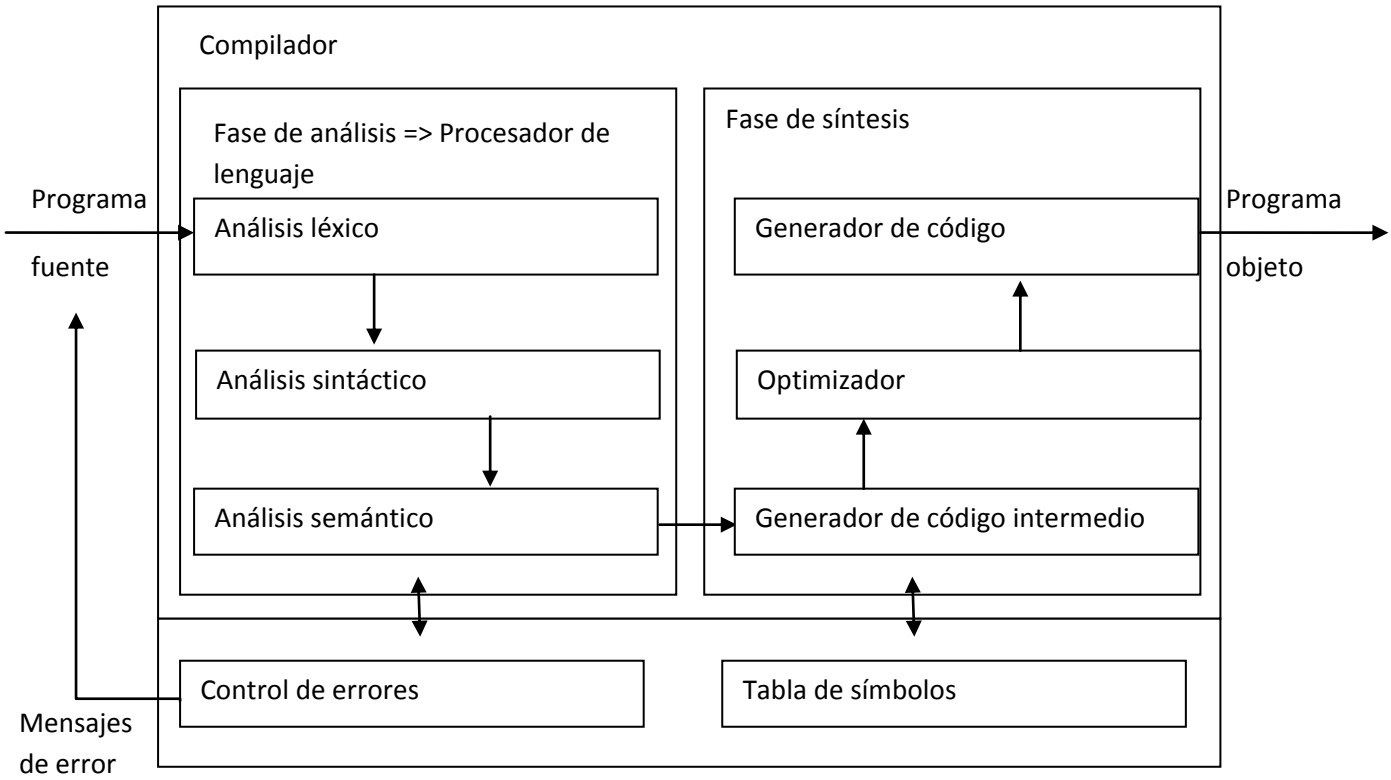
Procesadores del lenguaje

Pau Arlandis Martínez

Introducción

Un compilador es un programa que traduce un lenguaje (código fuente de otro programa) de código en otro texto en lenguaje objeto.



Este es el esquema básico de un compilador, veámoslo por partes:

Fase de análisis

En la fase de análisis se estudia el lenguaje del programa fuente, errores, elementos de interés… Esta fase también se denomina procesador del lenguaje y es lo que vamos a ver en esta asignatura. La fase de análisis se describe en tres subfases:

Análisis léxico

Se encarga de analizar los elementos más pequeños de un texto que tienen sentido por sí mismas.

Análisis sintáctico

Se encarga de analizar la estructura de un texto.

Análisis semántico

Se encarga de estudiar el significado de todos los elementos dentro del texto.

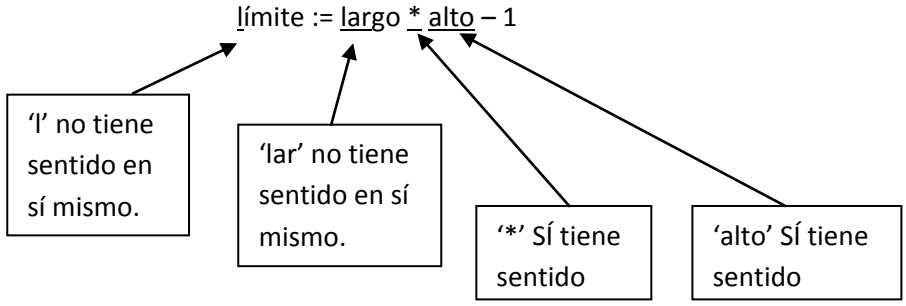
Ejemplo

Texto Fuente

límite := largo * alto – 1

Analizador léxico

El analizador léxico estudia carácter a carácter conociendo las estructuras mínimas que tienen sentido por sí mismas. A estas se las denomina *Tokens.*



Texto resultado

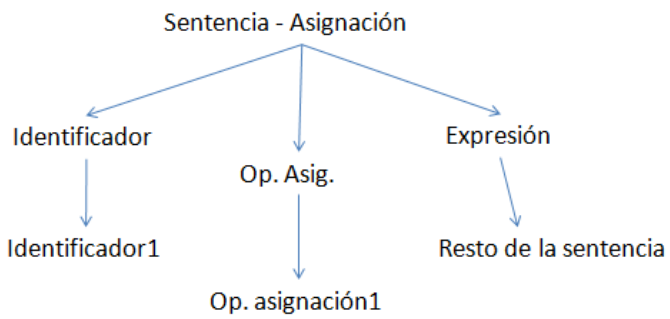
Identificador1/op.asignación1/identificador2/op. aritmética1/identificador3/op.aritmético2/constante_entera1/

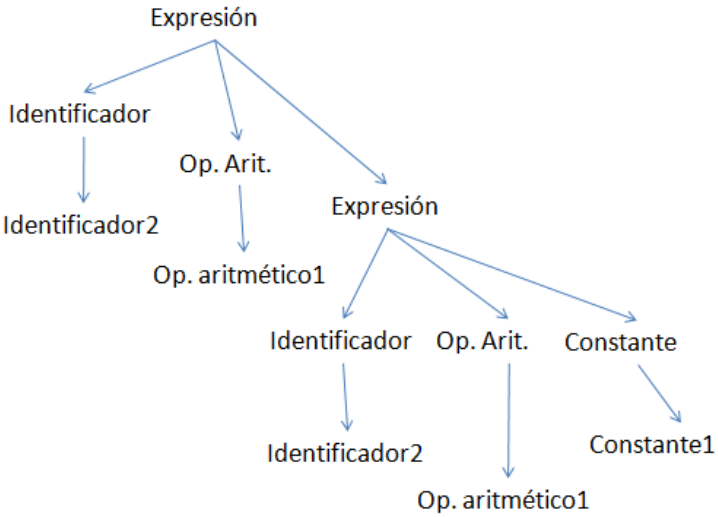
A grandes rasgos este sería el resultado del analizador. Cada *token* necesitaría más información pero eso lo veremos más adelante.

Analizador sintáctico

Estudia la estructura de la secuencia de *tokens* para conocer si es una estructura permitida o no lo es.

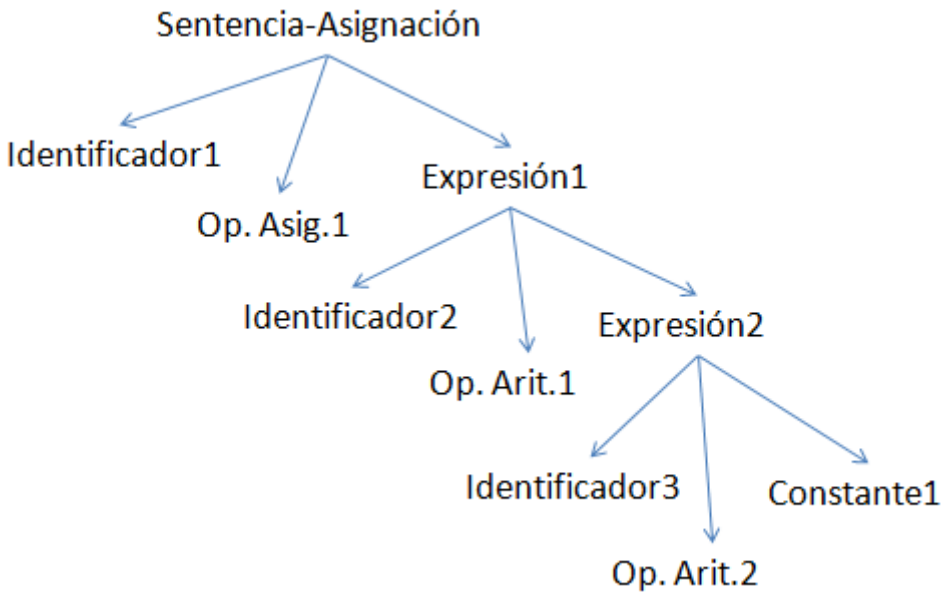
En el ejemplo:





Árbol resultado

Resulta entonces en un árbol sintáctico donde se detalla la estructura de la sentencia:

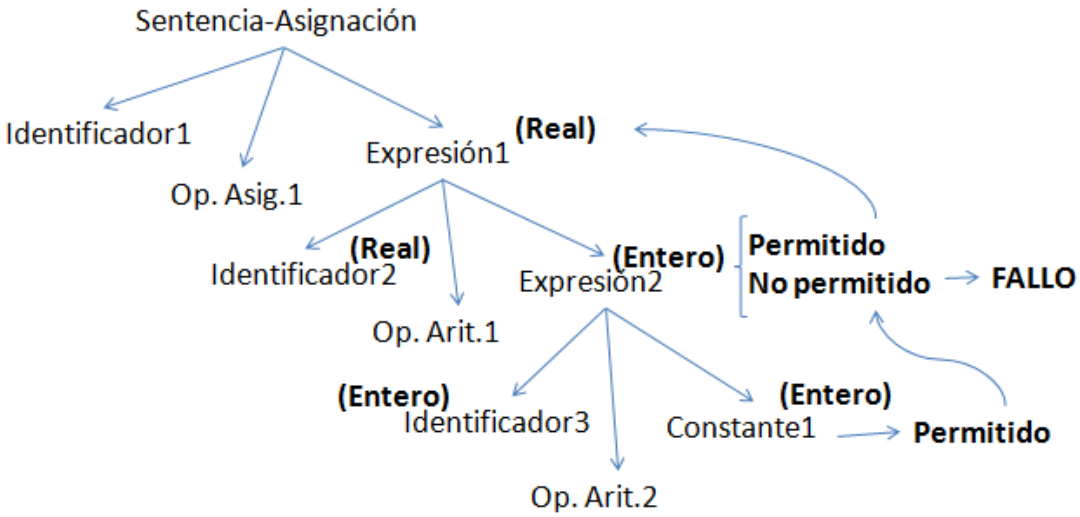


Este árbol es la entrada del analizador semántico que estudia si la sentencia (estructuralmente correcta) tiene sentido en el lenguaje.

Analizador Semántico

Este analizador hace anotaciones en el árbol sintáctico comprobando tipos, variables definidas,…

Por ejemplo:



Control de errores

Cada fase de análisis tiene sus propios errores:

- **Errores léxicos**: Se utilizan operadores inexistentes o palabras mal formadas → *limi?te , */*
- **Errores sintácticos**: Se estructuran mal los *tokens*, de forma incorrecta o confusa → *id1 op.asig id2 cte_entera*
- **Errores semánticos**: La estructura de *tokens* elegida es correcta pero no tiene sentido → $id1^{(Entero)}$ op.asig $expr^{(real)}$

Todos estos errores se analizan en el control de errores.

Tabla de símbolos

La tabla de símbolos es una colección de símbolos que utiliza nuestro texto para almacenar información de cada *token*. El procesador del lenguaje interactúa con ella para almacenar y consultar su contenido.

Estos 5 módulos de la fase de análisis son, básicamente, los temas y el contenido de la asignatura. Para completar la información, introduzcamos también el contenido de la fase de síntesis.

Fase de síntesis

Una vez resuelta sin errores la fase de análisis comienza la fase de síntesis que será la encargada de construir el texto objeto del texto fuente que hemos utilizado. La fase de síntesis se subdivide en tres módulos:

Generador de código intermedio

Este es el primer paso, su misión consiste en facilitar la traducción del lenguaje fuente al lenguaje objeto traduciendo el primero a un lenguaje intermedio. Utiliza el árbol anotado de la fase de análisis.

Su otra misión es muy útil para reutilizar módulos ya que podemos traducir muchos lenguajes al mismo lenguaje intermedio.

En el ejemplo:

C.I → temp1 := id2*id3

temp2:= 1

temp3:=temp1-temp2

id1:=temp1

Optimizador

El lenguaje resultante del generador de código intermedio (CI) suele ser demasiado redundante y poco eficiente ya que traducen paso a paso el árbol semántico sin fijarse en

pasos anteriores. Por ello es necesario el optimizador, para convertir el CI en un lenguaje eficiente:

O.C. → temp1 := id2*id3

id1 := temp1 – 1

Generador de código

El último paso del proceso de síntesis es muy similar al primero, el generador de código se encarga de traducir el CI optimizado resultado del proceso anterior, que es más fácil que el lenguaje inicial, en código ensamblador de muy bajo nivel.

En el ejemplo:

G.C. → move id2, r1

mul id3, r1

sub #1, r1

move r1, id1

Lenguajes y gramáticas

Repaso de la teoría de lenguajes

- **Alfabeto:** Conjunto finito de símbolos. *(a-z, A-Z)*
  - ***String* o cadena:** Cualquier secuencia de símbolos de un alfabeto. *hola*
- **Longitud:** Número de caracteres de una cadena. *hola → 4*
- **Cadena vacía:** Cadena con longitud 0, denominada λ.
- **Lenguaje:** Conjunto de cadenas válidas formadas por un determinado alfabeto.
  - **Válido:** Cumple unas determinadas reglas.
- **Gramática:** Especificación precisa y formal de todas las cadenas válidas para un determinado alfabeto. Definen y especifican un lenguaje. Se denomina como una 4-tupla (N, T, P, S) donde:
  - **N:** Símbolos no terminales (símbolos auxiliares)
  - **T:** Símbolos terminales (alfabeto del lenguaje). Una cadena solo puede estar compuesta de símbolos terminales.
  - **P:** Reglas de producción. Antecedente → Consecuente.
  - **S:** Elemento de N, llamado axioma, que es el generador de todas las cadenas, es el primer antecedente.

Ejemplo: $G1(N=\{A\}; T=\{0,1\}; P \rightarrow \left\{\begin{matrix} A \rightarrow 0A \\ A \rightarrow 1A \\ A \rightarrow 0 \end{matrix}\right\}; S=\{A\})$

A → 0

A → 0A → 00

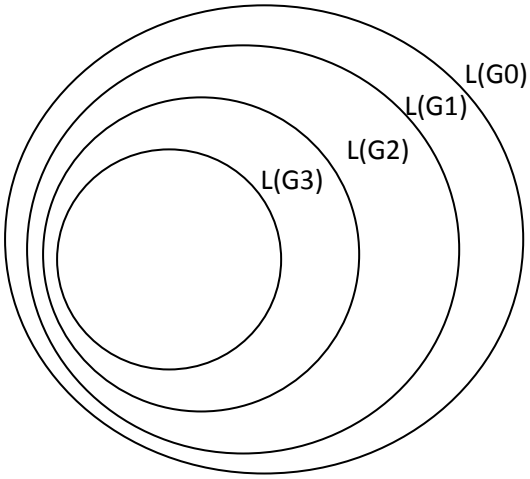
A → 1A → 10A → 100

Entonces:
L(G1) = { C/ A →*C} = {0,1}*0

Jerarquía de Chomsky

- **Tipo 0 (sin restricciones)** = Las reglas de producción pueden ser de cualquier tipo. $\alpha \rightarrow \beta$ $\alpha,\beta \in \{N \cup T\}^*$
- **Tipo 1 (dependientes del contexto)** = $\alpha A \beta \rightarrow \alpha \gamma \beta$ $A \in N$ $\gamma \in \{N \cup T\}^*$
- **Tipo 2 (Independientes del contexto)** = $A \rightarrow \gamma$
- **Tipo 3 (gramáticas regulares)** = $A \rightarrow aB$; $A \rightarrow a$ $a \in T$ $B \in N$

} Nos interesan en esta asignatura.



Las gramáticas de tipo 2 las usaremos en el análisis sintáctico.

Para el análisis léxico utilizaremos gramáticas de tipo 3 (regulares)

R. por la izquierda

B → Bα Ejemplo: B→B1

B→B1→B11

Existen analizadores sintácticos incompatibles con esta recursividad por eso debemos crear una gramática equivalente sin recursividad por la izq.

Eliminación Recursividad por la izquierda

A → Aα
A → β

A → B

Mantenemos las producciones normales

A → β
A'→α

A → βA'
A'→αA' ←Tras β hay tantos α como se quiera

Introducimos un símbolo N auxiliar y se elimina la recursividad por la izquierda.

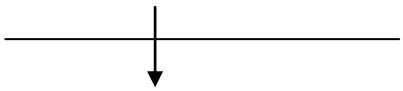
β
Aα → βα
Aαα → βαα
Aααα → βααα

β
βA' → βα
βA' → βαA' → βαα
βA' → βαA' → βααA' → βααα

Factorizar por la izquierda

A→ αβ A → 1B Existen

A → αϒ A → 1C



A → αA'
A' → β
A' → ϒ

Gramática ambigua

Una gramática es ambigua si existen dos formas diferentes de generar una misma sentencia, una misma cadena. Los procesadores del lenguaje no son compatibles con las G. ambiguas. Ejemplo:

1. S → if E then S
2. S → if E then S else S
3. S → P

y en el lenguaje aparece:

if e1 then if e2 then P1 else P2

de forma:

S $\rightarrow^{1}$ if e1 then S $\rightarrow^{2}$ if e1 then if e2 then P1 else S $\rightarrow^{3}$ if e1 then if e2 then P1 else P2

pero:

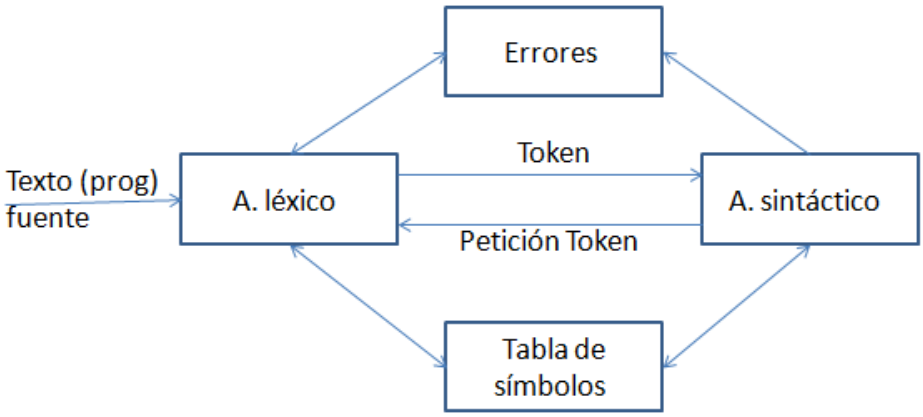
S $\rightarrow^{2}$ if e1 then S else S $\rightarrow^{1}$ if e1 then if e2 then S else S $\rightarrow^{3}$ if e1 then if e2 then P1 else S $\rightarrow^{3}$ if e1 then if e2 then P1 else P2

Hemos llegado a la misma cadena mediante dos caminos generadores. Esta es una gramática ambigua. Ningún analizador sintáctico permite una gramática ambigua ya que esta generaría distintos árboles y, por tanto, distintos códigos objeto.

Ejercicio

E → E + T
E → T
T → T*F
F → a
F → (E)
T → F
Es una gramática recursiva por la izquierda. Elimina dicha recursividad.

Análisis léxico



El analizador léxico simplifica el trabajo al Analizador sintáctico agrupando todas aquellas cadenas que tienen la misma estructura sintáctica. Por ejemplo, si recibe la cadena *límite* el A. léxico debe saber que esto es un **identificador** a nivel sintáctico y así con cada uno de los tokens.

Funciones del Analizador léxico

- Manejar el fichero del texto fuente (TF).
- Leer los caracteres del TF.
- Enviar al Analizador sintáctico un token.
- Eliminar del TF todo lo que sea superfluo. Por ejemplo: comentarios, espacios en blanco, saltos de línea…
- Detectar errores léxicos. Es decir, secuencias de caracteres que no corresponden a ningún token. Por ejemplo:
  - 10,00,0 → Número mal formado
  - 7a8 → Error léxico

  } Imposibilidad de formar un token
  - a**ñ**o, al**á** → Caracteres incorrectos en la mayoría de compiladores.
- Relacionar los errores con su posición en el TF.
- Pre procesar el TF (macros)
- Interactuar con la tabla de símbolos.

Definiciones

Token (componente léxico)

Cada uno de los elementos del lenguaje que estamos identificando que tienen sentido o significado por sí mismos. Es un conjunto mínimo de caracteres. Ejemplo:

limite := 100

Es un solo *token* ya que ':' o '=' no tienen sentido aquí en sí mismas.

if a = 3 then

Este '=' sí tiene sentido aunque vaya solo.

Todo *token* tiene algunos componentes importantes:

Lexema

Es el conjunto de caracteres que forman un *token*. Ejemplo:

límite (6 caracteres) en un identificador.

Patrón

La regla que describe como un determinado lexema se asocia a un determinado *token*.
Ejemplo:

"Secuencia alfanumérica (sin espacios ni signos de puntuación) que empieza por una letra" en un identificador.

Tipos de *token*

Existen algunos tipos básicos que aparecen de forma habitual:

- Identificador
- Número
  - Número real
  - Número complejo
  - Número entero
- Operadores
  - Asig. *:=*
  - Relac. *<, >, =, <=,…*
  - Aritm. *+, -, /, *, **,…*
- Puntuación → *; , {, }, (,), …* (Aunque hay que saber que cada uno de estos símbolos serian *tokens* distintos ya que se diferencian a nivel sintáctico.)
- Palabras clave → *If, Do, While,…* Como antes, aunque las hemos agrupado juntos, lo normal es que cada palabra clave forme un *token.*

Aunque existen muchos más.

Un Ejemplo

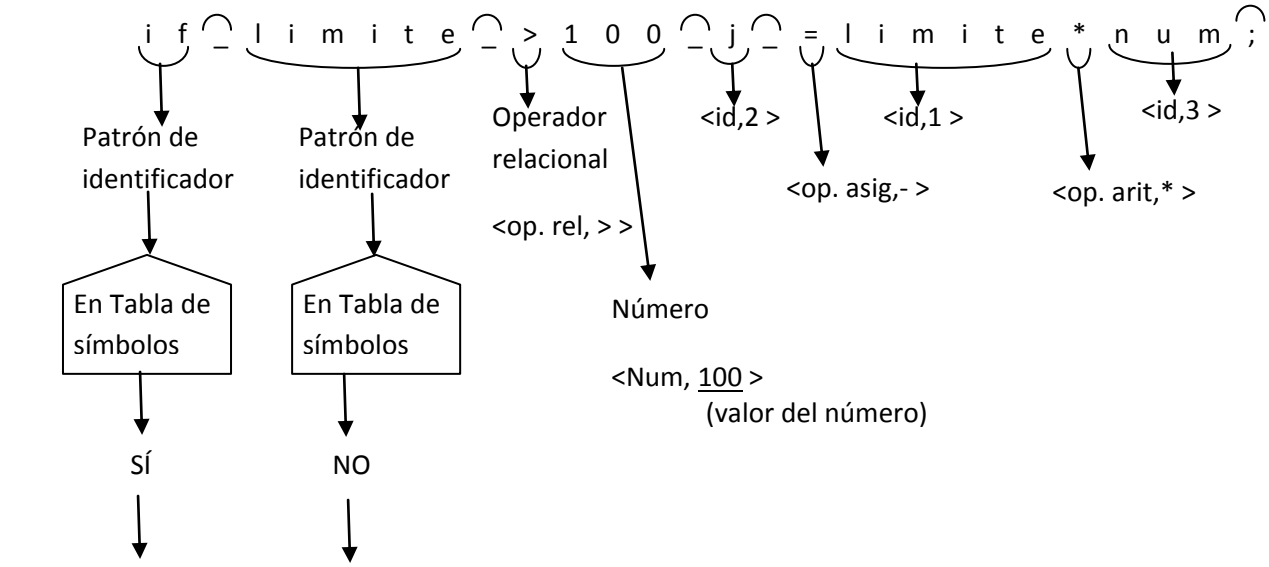
a + b * C $\xrightarrow[\text{léxico}]{\text{Analizador}}$ id1/op.aritm1/id2/op.aritm2/id3 $\xleftarrow[\text{léxico}]{\text{Analizador}}$ j – i / d

Como puede verse, ambas sentencias producen el mismo listado de *tokens* a pesar de ser muy diferentes. Si se enviaran al Analizador sintáctico se perdería la información de esa sentencia. Por eso debemos enviar más información.

Entonces, el *token* lleva información adicional. Un *token* será una tupla con la siguiente estructura:

Token = (Tipo de *token*, Información adicional)

Ejemplo de funcionamiento



Palabra clave

<if, ->
Con '-' se precisa que no es necesaria más información

Se coloca en la tabla de símbolos como identificador1

<id, 1>
Posición en TS

Tabla de símbolos

Pos	Lexema	Tipo	…
1	Límite		
2	J		
3	Num		
…	…	…	…

Diseño léxico

Todo analizador léxico dependerá del lenguaje sobre el que se va a diseñar. Cualquier diseño sigue un número determinado de pasos:

1. Determinar que *tokens* voy a tener. Paso fundamental.
2. Construir la gramática que identifica estos *tokens* (gramática tipo 3, regular).
3. Construir el diagrama de transiciones de un AFD.
4. Incorporar acciones semánticas. Cualquier otra cosa que haga el analizador que no sea encontrar los *tokens* se realiza mediante acciones semánticas (interactuar con la TS, traducir valores numéricos, etc).

5. Tratar errores (incluyendo mensajes de error).
6. Implementar el AFD.

Ejemplo

Lenguaje → Secuencia de identificadores separada por blancos. El patrón de los identificadores es el descrito anteriormente.

Por ejemplo:

límite_j_ind13

hola_estoy____aqui

Paso 1

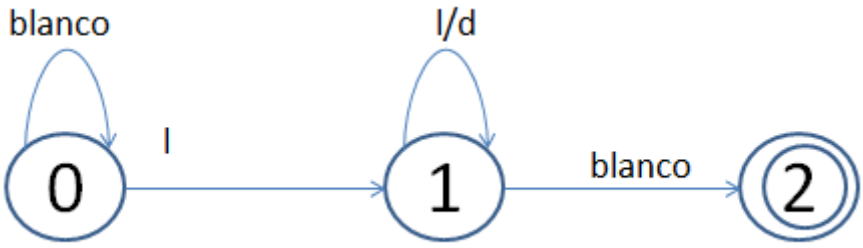
Tokens: <id, pos_TS>

Paso 2

<N, T, P, S> → P: { S → lA (l es cualquier letra l = a-z, A-Z)
A → lA / dA (d es cualquier dígito d=0-9)
A → λ }

Entonces: <{S, A}, {l, d}, P, {S}>

Paso 3



Paso 4

Acciones semánticas

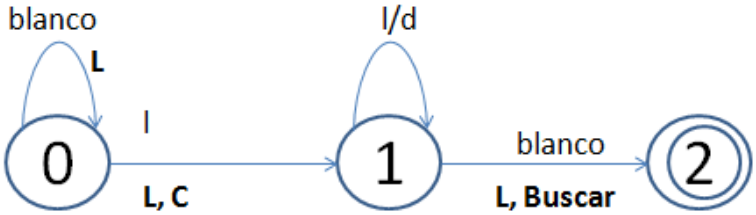
L : leer siguiente carácter ← Trivial, pero necesario especificar.

C': lexema es la l

C : lexema es el lexema o la concatenación leída l/d

Buscar: If pos := Buscar (lex, TS) then existe_token (id, pos)
Else emite_token (id, pos = inserter TS(lexema))

Se anotan en el diagrama del paso 3 en la correspondiente transición:



Paso 5

Llegados a este punto definiremos que cualquier cosa que no contemple el autómata es un error, sin necesidad de especificarlo de forma explícita. Sin embargo, habría que definir los

Ejercicio

Hacer un diseño igual que este para el lenguaje:

Secuencias de n^{os} reales separados por blancos sabiendo que:

nº real = entero[, decimal]

Ejemplo de analizador léxico 1

Paso 1: Tokens

- := → <op.asig, - >
- +, * → <op. arit, +>, <op. arit, *>
- > → <op. rel, > >
- identificador → < id, pos_TS>
- $n^{\underline{os}}$ enteros →<nume, valor>

Paso 2: Gramática regular

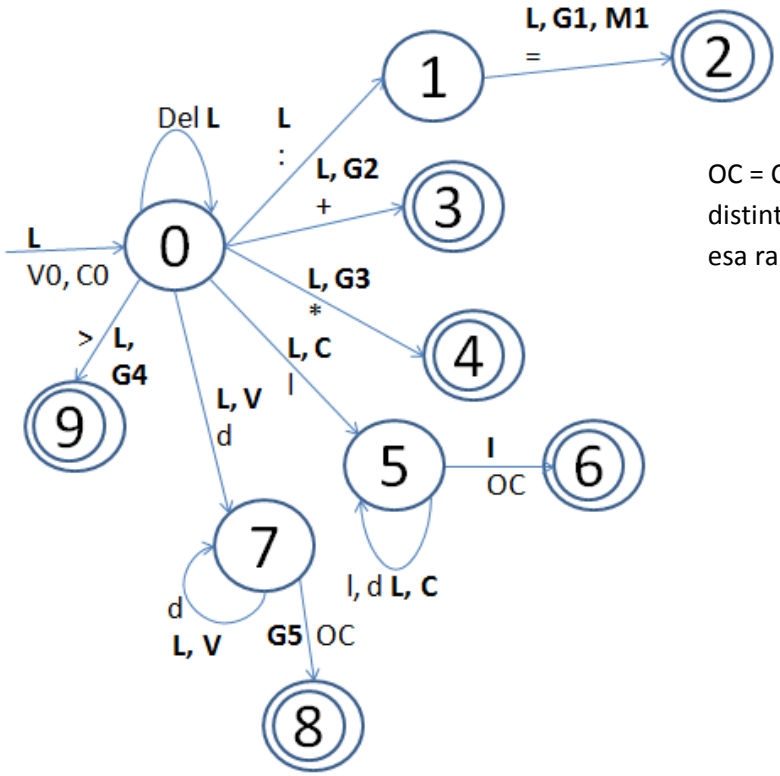
<N, T, P, S>

P: { S → :A/+/*/>/lB/dC/del S → Del S no forma parte del proceso de la gramática, ayuda al autómata.
A → =
B → lB/dB/λ
C → dC/λ

l = letra= a-z, A-Z

d= dígito = 0-9

Paso 3: Autómata finito



OC = Cualquier otro carácter distinto de los ya especificados en esa rama

Paso 4: Acciones semánticas

- **L** = leer siguiente carácter.
- **C0 =** lexema = " " (iniciamos variable)
- **C** = concat (lexema, l/d)
- **I** = posición := buscar (lexema, TS)
  If posición = null then posición := insertar (lexema, TS)
  Generar token (id, posición)
- **V0** = valor = 0 (iniciamos variable)
- **V** = valor * 10 + d = valor ('d'=valor del dígito)
- **G1 =** Generar token (op.asig, -)
- **G2 =** Generar token (op. arit, +)
- **G3 =** Generar token (op. arit, *)
- **G4 =** Generar token (op. rel, >)
- **G5=** Generar token (num_l, valor)

Paso 5: Errores

Los mensajes de error se crean y envían desde el módulo de control de errores. Un analizador léxico solo tendría que enviar a dicho módulo un código de error para cada operación que no se contemple en el autómata finito.

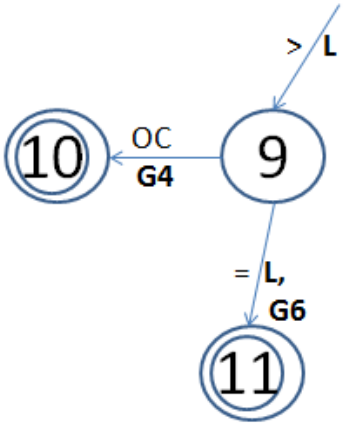
El único error que podría darse en este ejemplo sería **M1** cuando después de ':' no se lee un '='.

Hay dos formas básicas de manejar un error, descubrirlo y terminar (que es la forma fácil) o volver al principio para subsanar el error.

> **Ejercicio**
> ¿Qué cambios habría que realizar en los puntos 3 y 4 para tratar palabras reservadas?

Añadidos al ejemplo

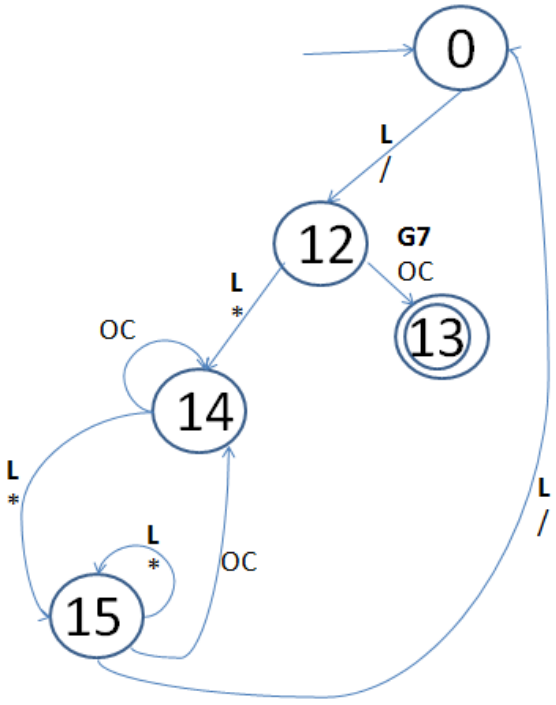
Un cambio que podríamos realizar es añadir un op. relacional nuevo: ">=". En el autómata modificamos la rama del op. relacional (estado 9), de la siguiente forma:



Y añadir la acción semántica **G6N**=Generar token(op.rel., >=)

Ahora vamos a seguir añadiendo más características típicas de un lenguaje de programación a este modelo. Por ejemplo:

- Comentarios: */* esto es un comentario *//*
- Cadenas de caracteres: *"esto es // una /"cadena= → esto es /una "cadena*



Con la doble barra añadimos una barra a la cadena y una barra con comillas sirve para añadir comillas a la cadena.

Comentarios

Para los comentarios debemos añadir al lenguaje '/'

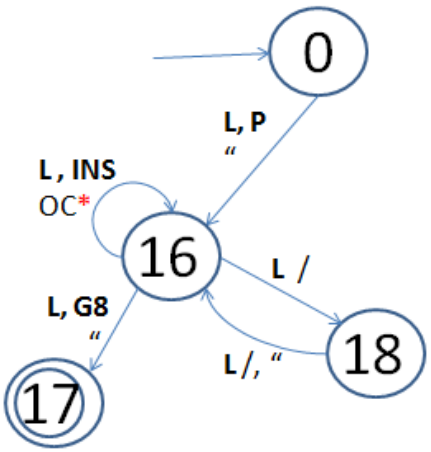
Los comentarios no generan tokens, simplemente se obvian y se vuelve al inicio para encontrar tokens.

Añadimos la acción semántica **G7**=Generar token (op. arit., /) para, ya

que estamos, añadir el operador división.

Cadena

La cadena es algo diferente porque sí genera un token: token cadena <cadena, puntero a memoria> Se almacena en memoria debido a su tamaño, demasiado grande para la TS. Debemos añadir al lenguaje |“| (comillas dobles)



Se añaden las Acciones semánticas:

G8 = Generar token (cadena, punt. mem.)

P = Reserva memoria.

INS = Concatena el contenido de la cadena.

*Salvo “ y /

Ejemplo de analizador léxico 2:

Paso1

Analizador de números reales.

$dig^{+}[.dig^{+}][e[signa]dig^{+}]$

- <entero, valor>
- <real, valor>

Paso 2

Gramática:

S → dN (te aseguras que siempre hay un dígito)

N → dN/λ/.D/eE (bucle para crear entero/fin de la parte entera sin decimales y sin exponente/decimales tras la parte entera/exponente tras la parte entera)

D → dD’ (un dígito tras el punto decimal como mínimo)

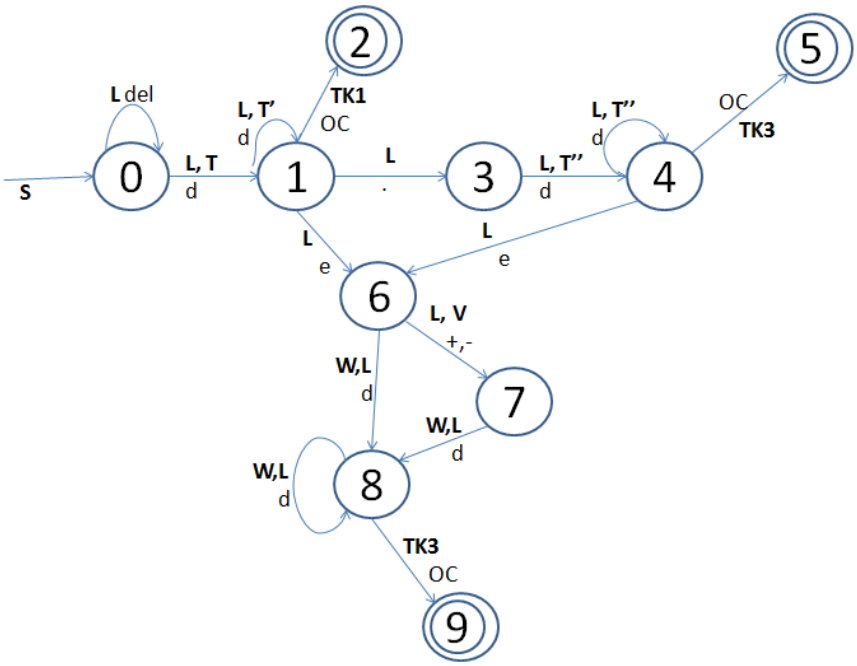
D’ → dD’/λ/eE (bucle para crear más decimales/fin de la parte decimal sin exponente/exponente tras la parte decimal)

E → +E’/-E’/dE’’ (exponente positivo/exponente negativo/sin signo)

E’ → dE’’ (aseguras un dígito tras el signo)

E’’ → dE’’/λ (bucle para crear más exponente/fin del exponente)

Paso 3



Paso 4

Acc. semánticas

L = Leer siguiente carácter (Iría en todas las transiciones excepto en OC)

S : Iniciamos variables.

- num = 0
- exp = 0
- ndec = 0
- sig = +1
- Tipo = null

T = num = d

T’ = num = num*10+d

T’’ = num = num*10+d
ndec = ndec+1 (aumenta el número de posiciones decimales)
por ejemplo:

→12,57 num=1257
ndec=2

TK2 = valor = $num/10^{dec}$
Generar token (real, valor)

TK3 = valor = num * $10^{(exp + sig) - ndec}$
Generar token (real, valor)

V = if "-" then sig = -1

W = exp = exp*10+d

Ejemplo de Paso 1

Tenemos la siguiente gramática:

S → Switch (E) do {C} / break / id := E ; / λ

L → case N:SL / default : S / λ

E → id |N|-E / (E)

} Reglas sintácticas

N → cte / -cte

id → id l / id d / l

cte → cte d / d

} Reglas léxicas

Haya los tokens necesarios:

- <id, pos_TS>
- <cte, valor>
- <pal_clave, código>
- <(, ->
- <),->
- <{, ->
- <}, ->
- <; ,->
- <: , ->
- <:=, ->
- <- ,-> signo menos

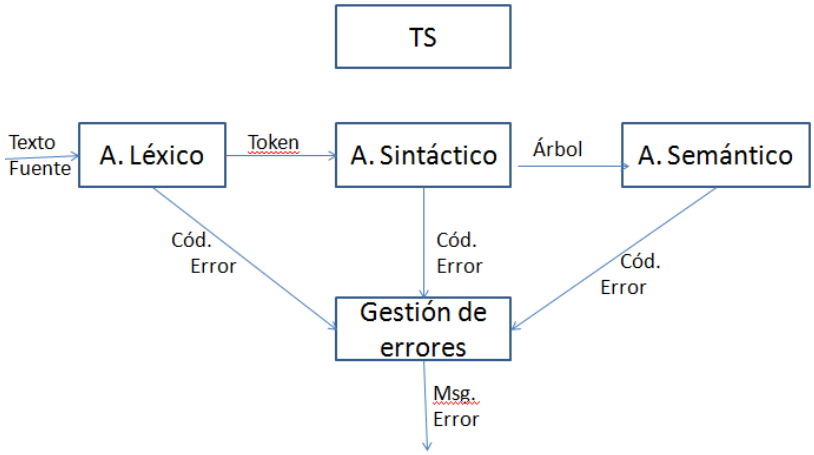
Ejercicio

Dado el siguiente texto en xml:

```
<tfc>
      <autor> Luis Gomez </autor>
      <fecha_nac> 18/05/85 </fecha_nac>
      <título> Diseño de An. Léxico </titulo>
      <tutor> Marian Heras </tutor>
      <nota> Sobresaliente 9 </nota>
</tfc>
```

Se pide analizador léxico para pasar este código a la base de datos. Paso a paso

Errores



Cómo tratar los errores

- Obviarlos, que no envíe ningún mensaje. → No se debe utilizar.
- Señalar el error y parar. Es la forma válida más sencilla, pero ineficiente ya que si hay varios errores tendremos que reiniciar el procesador tras solucionar cada uno.
- Señalar el error y continuar buscando más errores (sin generar código). Es más eficiente que el anterior pero tiene el problema de que no siempre un procesador se recupera de un error y podría arrastrarlo y que se generen más errores.
- Señalar el error y tratar de corregirlo. La mejor forma, pero la más compleja.

Errores típicos

Análisis léxico

- Caracteres que no pertenecen a los terminales. *Año*
- Identificadores mal formados. *3ab*
- Constantes mal escritas. *3.0.4*
- Constantes fuera de límite. *12975432*

Análisis sintáctico

- Símbolos no permitidos en esa posición. *If… then **3** Begin…*
- Ausencia de delimitadores. *i=3__f=4;*
- Palabras reservadas mal formadas. *if…Dhen…*

Análisis semántico

- Incompatibilidades de tipo en expresiones/asignaciones. *entero = entero*3,0*
- Inconsistencia de tipo entre parámetros formales y actuales.
- Identificadores no declarados o definidos múltiplos de un identificador.

Mensajes de error

Los mensajes de error deben seguir una serie de características:

- Expresados en términos del usuario.

- Localizar los errores adecuadamente.
- Completos y legibles.
- Amables y respetuosos.

Tabla de símbolos

La tabla de símbolos (TS) es el lugar en el que se guarda, de forma ordenada, todos los identificadores que existen en el texto fuente. La TS es una estructura de datos que está presente mientras se ejecuta el compilador, entonces, obviamente, no existe cuando se ejecuta el programa. Esta es una cuestión crítica.

Tipos de información

- Lexema. La secuencia de caracteres que representa al identificador en el texto fuente.
- Tipo del identificador.
- Índice, aunque es más una característica de la propia TS. Suele almacenarse en una estructura dinámica.
- Alcance.
- Dimensiones.
- Argumentos (procedimiento o función).
- Tipo de valor devuelto (función).
- Dirección de memoria estática, donde se almacena el valor de la variable cuando se ejecute el programa. Enlace. El propio valor nunca se guarda en la tabla ya que es necesario en ejecución, cuando la TS ya no existe.

La TS no es una tabla estática ya que la cantidad de memoria que se necesita para guardar cada identificador es variable, depende del número de identificadores y la cantidad de información que tengan. La TS es, por tanto, una tabla dinámica.

Organización de la TS

Imaginemos los siguientes identificadores: A, Aux, I, Cont, X

Podemos organizarlos en diferentes TS:

TS Lineal

Se introducen de forma lineal en la tabla.

Iden.	Lexema	Tipo	...
1	A	...	...
2	Aux	...	...
3	I	...	...
4	Cont	...	...
5	X	...	...

Para meter el identificador el corte es trivial, pero no así para buscar dentro de la TS ya que, en el caso peor, tienes que recorrer toda la tabla. Durante la compilación es mucho más usada la opción de buscar que la de insertar, por tanto esta organización es muy ineficiente.

TS ordenada

Iden.	Lexema	...
1	A	...
2	Aux	...
3	I	...
4		

Cont > I

Iden.	Lexema	...
1	A	...
2	Aux	...
3	Cont	...
4	I	...

A > Aux > I

A> Aux > Cont> I

TS hash

En este caso la tabla se ordena según una función hash que le asigna un índice.

Función → Número de caracteres (función hash un poco inútil).

Iden.	Lexema	...
1	A	...
2		
3	Aux	...

A → 1

Aux → 3

I → 1

Ya está siendo usada (la función hash no es perfecta)

Para evitar esto existen varias soluciones:

1. El identificador se coloca en la siguiente posición vacía ← No es la mejor.
2. Se crea una tabla de desbordamiento a la que las posiciones con varios identificadores se dirigen cuando se busca o se inserta un identificador de esa posición.

Iden.	Lexema	...	Desbordamiento
1	A	...	
2			
3	Aux	...	

Lexema	Tipo	...	Desbordamiento
I	...	...	
X	...		
Con	...	...	

Con = 3

Solo se enlazan los identificadores con igual índice.

La evolución natural de esta solución es que la TS sea una lista de índices que enlacen con el primer identificador de ese índice en la tabla de desbordamiento.

Iden.	Des.
1	
2	
3	
4	

Lexema	...	Desbordamiento
A	...	
Aux	...	
I	...	
X	...	
Con	...	
J	...	
Cont	...	

1

1

3

1

Manejo de errores con la TS

```
1     Programa básico
2     Var a:integer
3     Procedure suma (a:integer, b:integer)
4           Begin
5                 a:=a+b
6           End
7     End
8     Begin
9           Read (a, b)
10          Print (suma(a,b))
11    End
```

Problema 1 En este caso existen dos variables 'a', una dentro del *Procedure* y otra global. El compilador considera que esta variable es la más cercana, en este caso la que está dentro del proceso. Debe solucionarse también con la TS.

Problema 2 Identificador no declarado. En este caso se debe saber esto gracias a la TS y conociendo el alcance de dicho identificador.

Solución del 1er problema

Doble variable en el texto. Para solucionarlo la opción más común es crear **otra** TS solo para el alcance más pequeño de una de las variables, en este caso, el procedimiento suma.

TS Global	
a	Integer

TS Suma	
a	Integer
b	Integer

Durante el proceso de compilación se utilizará la TS de la zona en la que estemos cada vez. Hasta la línea 3 se utiliza la TS actual es TS Global. Al encontrar la palabra *procedure* (por ejemplo, hay que saber que palabras clave crean una TS) crearemos una nueva TS con el nombre de ese proceso y la TS actual pasa a ser dicha TS.

Esta es la forma de conocer el alcance de un identificador → Solución 2º problema.

Ejercicio resuelto

Dado este programa

```
int a,b;  /*globales */

void proc1 (int *x)
{
      int b;
      b = 3;
      *x = b * 2 + a;
}

boolean fun (int x)
{
      int a;
      a = 2;
      return (u % a) ==
b;
}

void proc2 (int d)
{
      int u;
      boolean v;
      u = a + d;
      v = fun (u);
}

void main ()
{
      a = 1;
      proc1(&a);
      proc2(&a);
      fun (b);
}
```

Normalmente ejercicios de este tipo piden determinar cómo avanza la TS.

Se definen durante la evolución:

- Tabla actual (TA) → Que indica que TS se desarrolla en ese momento.
- Definición/uso (D/U) → Que indica en que zona estamos, o en definición de variables o en uso de ellas.

Primero se crea la TS global.

Id.	Lexema	tipo	Valor Devuelto
1	a	Int.	-
2	b	Int.	-
3	proc1	Función	Void
4	fin	Función	Boolean

TS Global al final del ejercicio

Comienza el proceso, se inicia el analizador sintáctico.

TA → TS Global
D/U → D

Se lee primero <palClave, int> que es una palabra clave. Después lee 'a' <id, 1> busca en la tabla de símbolos y al no estar lo inserta (por estar en zona de definición de variables). Lee <, , -> y después <id, 2> que se inserta. Continúa leyendo <; , ->, el comentario se salta, y después <palClave, void>. A continuación de 'void' sabemos que comenzará una zona nueva de definición, entonces se crea la tabla de proceso. <id, (proc1)> → <id, 3> este *token* se define en la TS Global porque puede accederse desde cualquier lugar del programa.

Id.	Lexema	tipo	...
1	x	Int. *	...
2	b	Int.	...

TS proc1 al final del ejercicio.

Ahora es cuando se cambian los flags:

TA → TS proc1
D/U → D

Continuamos leyendo. Tras el paréntesis se lee 'x' <id, 1> y se coloca en la TS proc1, porque es la actual. Después se lee el símbolo llave y tras ella, se cambia a zona de definición (D/U → U). Al leerse la palabra clave 'int' se sabe que se va a definir una variable y se cambia a

zona de definición (D/U → D). Se añade el identificador 'b' y se vuelve a la zona de uso (D/U → U). Entonces se lee la variable 'a', al estar en zona de uso se busca en la TS actual (TS proc1) al no estar se busca en la TS Global, como está se envía el *token* <id, 1(TS Global)>. Cuando se llega al símbolo cierre de llave la TS proc1 se borra, ya no hace falta más y el flas TS pasa a ser TS Global.

Se continuaría de esta forma, creando a continuación la tabla fun para la función fun. Lo único destacable es que al leer el carácter 'u' en zona de uso en la TS fun y no estar en ninguna tabla saltaría un error.

Procesos anidados

Dado el ejemplo:

```
PROGRAM Ejemplo;
      VAR a, b : INTEGER;

PROCEDURE proc1 ( VAR  x: INTEGER);
      VAR b : INTEGER;
BEGIN
      b := 3;
      x := b * 2 + 4;
END;

PROCEDURE proc2 (d : INTEGER);
      VAR   u : INTEGER;
            v : BOOLEAN;
      FUNCTION fun (x : INTEGER) : BOOLEAN;
            VAR a : INTEGER;
      BEGIN
            a = 2;
            return ( u MOD a) = b;
      END;
BEGIN
      u := a + d;
      v : = fun (u);
END;
BEGIN
      a := 1;
      proc1 (a);
      proc2 (a);
      fun (b);
END;
```

Flags	Tabla actual	~~Global~~	~~Proc1~~	~~Proc1~~	~~Global~~	~~Proc2~~	~~Fun~~	~~Fun~~	~~Proc2~~	Proc2
		1	2	3	4	5	6	7	8	9
	Definición/ uso	~~def~~	~~def~~	~~uso~~	~~def~~	~~def~~	~~def~~	~~uso~~	~~def~~	uso

Histórico de la tabla de definición/uso a lo largo del ejercicio.

Paso 1

Tabla bloques

Bloque	Bloque padre	Puntero
Global	-	
Proc1	Global	
Proc2	Global	
Fun	Proc2	

TS Global

Ind.	lexema	tipo	Desplazamiento
1	Ejemplo	Program	-
2	a	Entero	0
3	b	Entero	2
4	Proc1	Procedure	-
5	Proc2	Procedure	-

Al leer el primer *Procedure* se crea un nuevo bloque anidado a TS global. Se pasa al paso 2.

Paso 2

TS Proc1

Ind.	lexema	tipo
1	x	Integer
2	b	Integer

Cuando se llega al primer BEGIN se ha terminado de construir la tabla Proc1 y se pasa a la zona de uso.

Al acabar (END) se elimina la TS Proc1 y también de la tabla de bloques. Y se pasa al paso 3

Paso 3

Cuando se lee el siguiente proceso se crea la tabla correspondiente.

TS Proc2

Iden.	lexema	tipo	Valor dev.
1	d	Int	-
2	u	Int	-
3	v	Boolean	-
4	Fun	Function	Boolean

El identificador Fun crea un bloque nuevo, hijo de Proc2. Se pasa al paso 4.

Paso 4

Iden.	lexema	tipo
1	x	Int
2	a	Int

'a' ya existe en otros bloques, pero al estar en zona de definición es como un lexema nuevo.

Cuando se pasa a zona de uso y se encuentra BEGIN se encuentra el carácter u, se busca en *fun*, al no estar se va al bloque padre de *fun*, *proc2,* donde sí existe, entonces se usa ese. Con 'b' pasa lo mismo pero hay que buscar en TS Global.

Al terminar esto, se borra la tabla *Fun*. Se pasa a la tabla padre y se continuaría.

Tratamiento de registros en TS

Esto será necesario para la práctica. Tenemos, por ejemplo, la siguiente estructura:

```
Struct Fecha
{int día, mes, año;
}
Fecha x;
```
← Registro de tipo fecha.

```
mimes = x.mes;
```
← Acceso a campos del registro.
son dos tokens

Tabla de símbolos actual

B	lexema	tipo	Tamaño reg.	campos
...	...	...	No importa lo que ya haya en la TS	
...	...	...		
n	Fecha	Registro	6	
n+1	x	Fecha	-	-

TS Fecha

	lexema	tipo	Despl.
1	Día	Int	0
2	Mes	Int	2
3	Año	Int	4

Al leer los diferentes campos que habrá en un registro. La mejor forma de representarlo es añadir una TS adicional con el identificador fecha que indexe los campos del registro.

Al cerrar el registro fecha la TS Fecha se da por finalizada y la suma de su tamaño (desplazamiento) será el tamaño del registro.

Lo siguiente que sucede es la creación de una fecha. Como siempre se añade a la TS de forma normal.

Entonces, ahora se quiere acceder al registro X, como siempre se busca los *tokens* para enviar.

<id, TS(n+1)>
<punto, -> → En este momento se cambia la TS actual a TS Fecha. Ya que se va a querer acceder a ella.
<id, TSFecha(2)> → Tras esto se vuelve a la TS General.

Con este tipo de organización de la TS se sabe siempre que tabla se está usando y se elimina la confusión, permitiendo doble asignación de variables, por ejemplo.

```
Array (8) of Fecha y;  Se crea un vector de 8 registros Fecha.
```

Se introduciría en la tabla de símbolos de forma corriente.

Análisis sintáctico

En el análisis sintáctico los elementos de trabajo son los tokens que le envía el A. léxico. Este módulo es el motor del procesador de lenguajes y es el que marca el ritmo de trabajo. Cuando lo necesita pide tokens al A. léxico y cuando tiene estructuras correctas las envía al A. Semántico.

El analizador sintáctico necesita una gramática más avanzada que el léxico. El léxico utilizaba una gramática tipo 3, el A. Sintáctico utilizará una gramática de tipo 2 (independiente del contexto), donde:

A → α AϵN
       αϵ(N U T)*
Es decir, que la única restricción es que a la izquierda haya un no terminal.

El resultado de trabajo en este módulo es un árbol sintáctico que comprenda si es correcto o no y donde estén los errores. Existen dos formas de construir el árbol:

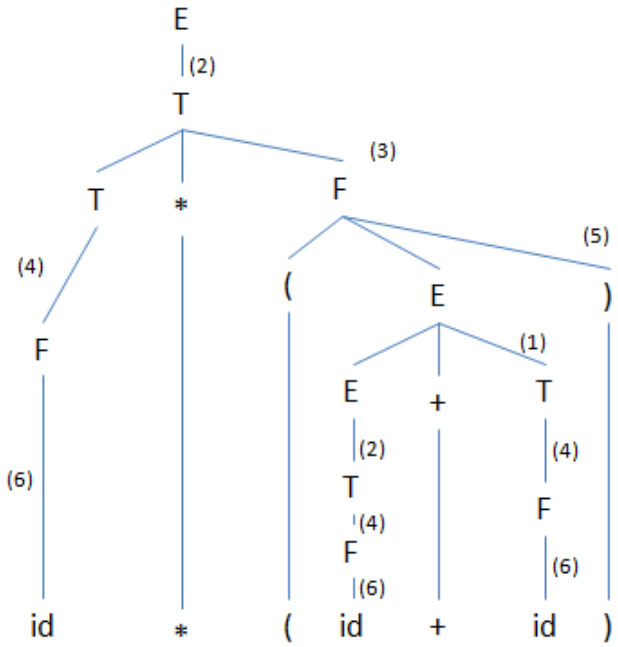
- Construcción ascendente → En esta asignatura: Analizador Sintáctico Ascendente LR
- Construcción descendente → En esta asignatura: Predictivo – recursivo. / LL o por tablas.

Veamos un ejemplo:

G : (E, N, T, P)
N = {E, T, F}
T = {+, *, (,), id}
P = E → E+T (1)
     E → T (2)
     T → T*F (3)
     T → F (4)
     F → (E) (5)
     F → id (6)

Dada la cadena:
id * (id + id)

Método descendente



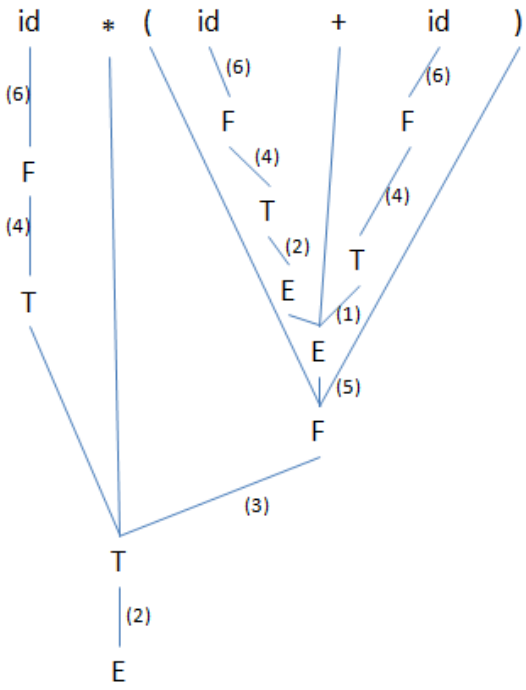
Siempre se expande el no terminal más a la izquierda, por convenio.

Esta es la forma gráfica de representar el árbol, existe también una manera de hacerlo más textual, solo con el número de las producciones efectuadas:

{2 3 4 6 5 1 2 4 6 4 6

Este método se denomina Parse o análisis a izquierdas debido a la forma de expandir.

Método descendente



6 4 6 4 2 6 4 1 5 3 2

Este se denomina análisis a derechas.

Método ascendente → arriba a derecha.

Método descendente → arriba a izquierda.

Gramática ambigua

1. E → E + E [id + id * id]
2. E → E * E
3. E → id

Análisis ascendente

<u>id</u> + id * id $\xrightarrow{3}$ E + <u>id</u> * id $\xrightarrow{3}$ <u>E + E * id</u> → E + <u>E * E</u> → <u>E + E</u> $\xrightarrow{1}$ E

pivote

Sería equivalente:

<u>id</u> + id * id $\xrightarrow{3}$ E + <u>id</u> * id $\xrightarrow{2}$ <u>E + E * id</u> $\xrightarrow{1}$ E * <u>id</u> $\xrightarrow{3}$ <u>E * E</u> $\xrightarrow{2}$ E

Una gramática ambigua es una gramática para la cual existe más de un árbol que la genere. No nos interesan las gramáticas ambiguas ya que queremos que el compilador siempre funcione de la misma manera.

Una gramática no ambigua de la anterior es:

E → T / E + T

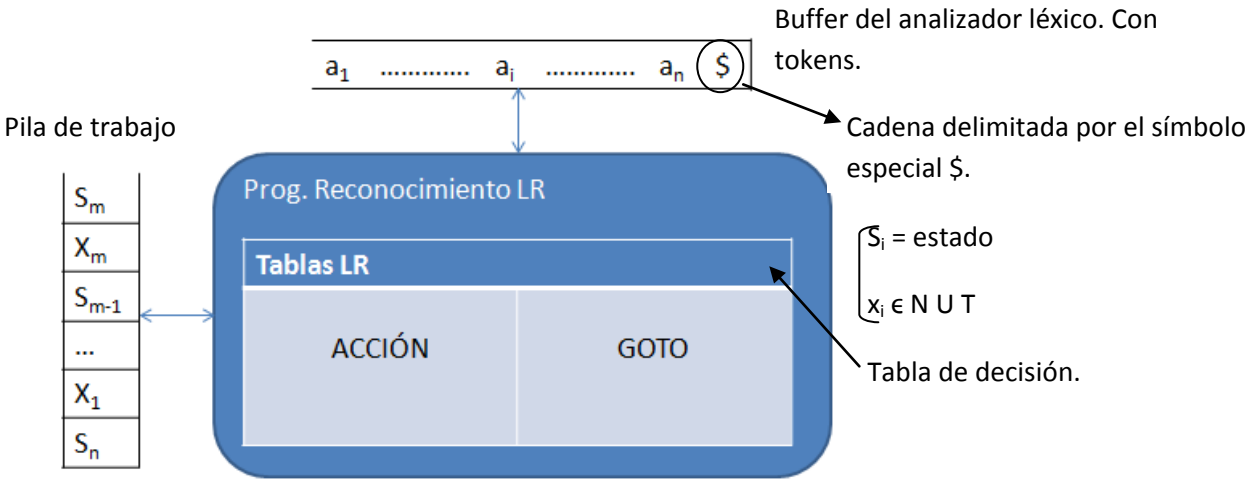
T → F / T * F

F → (E) / id

Existen formas de análisis que nos permiten manejar mejor la ambigüedad.

Análisis ascendente por reducción/desplazamiento: LR

Existen varios tipos de LR (SLR, LR canónico, LALR), nosotros vamos a utilizar el SLR. Este método es suficientemente general como para reconocer cualquier construcción que existe hoy en día en los lenguajes de programación. Detecta los errores lo antes posible.



El inconveniente que tiene es que es el método más engorroso de construir.

Tabla de decisión

Acción [Estado x token]

Acción (S_j , a_i) = { Reducir A → α
Desplazar y transitar al estado S_k
Error (Código error).
Aceptar.

Reducir

(S_0, x_1, …, x_m,S_m; a_1, …, a_n\$) ⟶ (S_0, x_1, …, x_{m-r}, S_{m-r}, A, S; a_1, …, a_n\$)

X_{m-r+1},…, X_m = α A → αB

S ← GOTO (S_{m-r}, A)

Desplazar

Significa pedirle un *token* al léxico y meterlo en la pila.

(S_0, x_1, …, x_m,S_m; a_i, …, a_n\$)

↘ (S_0, x_1, …, x_m,S_m, a_i, S_k; a_i, …, a_n\$) → S_k se introduce puesto que nunca puede haber un *token* en la cima de la pila.

Ejemplo de análisis sintáctico LR

E → E + T
E → T
T → T * F
T → F
F → (E)
F → id
Con la palabra → id * id

Tabla de decisión

	ACCIÓN						GOTO		
estado	id	+	*	(	)	$	E	T	F
S_0	D5			D4			1	2	3
S_1		D6				ACEPTAR			
S_2		R2	D7		R2	R2			
S_3		R4	R4		R4	R4			
S_4	D5			D4			8	2	3
S_5		R6	R6		R6	R6			
S_6	D5			D4				9	3
S_7	D5			D4					10
S_8		D6			D11				
S_9		R1	D7		R1	R1			
S_{10}		R3	R3		R3	R3			
S_{11}		R5	R5		R5	R5			

D = desplazar R = reducir

PILA	Cadena	ACCIÓN
S_0	~~id~~ * id $	D5
S_0~~id S_5~~	* id $	R6 (F→id)
S_0~~F S_3~~	* id $	R4(T→F)
$S_0 T S_2$	* id $	D7
$S_0 T S_2 * S_7$	~~id~~ $	D5
$S_0 T S_2 * S_7$~~id S_5~~	$	R6(F→id)
$S_0 T S_2 * S_7$~~F S_{10}~~	$	R3(T→T*F)
S_0~~T S_2~~	$	R2(E→T)
$S_0 E S_1$	$	ACEPTAR

A la hora de tomar decisiones se tiene en cuenta el estado y el *token* actual. Mirando la tabla de decisión: S_0 con id indica D5, desplazar al estado 5. Entonces se añade el *token* a la pila, se elimina de la cadena y como siempre debe haber un estado en la cima de la pila añadimos el estado 5.

Ahora tenemos estado 5 y * como *token,* eso nos da R6 (reducir por la regla 6) eso significa reducir por F→id, en la pila se eliminan tantos símbolos como el doble de elementos a la derecha de la regla, en este caso hay un símbolos (F →id), por tanto en la pila se eliminan dos símbolos (id, S_5) uno de los símbolos debe coincidir con el que está a la derecha de la regla, en este caso es correcto. Entonces en la pila se coloca el símbolo a la izquierda de la regla (S_0F) y para conocer qué estado hay que poner a continuación se va a la tabla GOTO con el estado en la pila (S_0) y el símbolo (F) entonces nos lleva al estado 3. Así vamos utilizando el algoritmo LR y poco a poco se llega a la situación de la pila final.

Construcción de tablas de decisión LR

Prefijo viable

Son todos los prefijos de una forma sentencial que no van más allá del pivote.

El pivote es la subcadena que vamos a tomar para reducir y que forma parte del camino correcto.

Dada la cadena:

a B c d e siendo cd el pivote y reduciendo por la regla A → cd quedaría:

a B A e y los prefijos viables serían:

a
aB
aBc
aBcd

La base de la construcción de las tablas de decisión LR es encontrar todos los prefijos viables de cualquier subcadena del lenguaje. Para ello utilizaremos un autómata reconocedor de prefijos viables.

Ítem

También llamado elemento.

Es cualquier regla de la gramática con un punto a la derecha de la regla.

E → E · + T
E → · T
F → id ·

Si yo quiero aplicar la regla que me indica el ítem necesito encontrar todos los elementos a la derecha del punto, es decir:

F → · id /El A. léxico nos pasa un identificador→ F → id ·
Para poder aplicar esta regla necesito un identificador.
Entonces ya puedo aplicar la regla.

Dos funciones para el manejo de ítems

Función cierre

Dado una G, I = conjunto de ítems válido.

Cierre (I) : I ⊆ Cierre (I)
Si A → α ·B β ϵ Cierre (I) => ∀B → ϒϵG, B→·ϒ ϵ Cierre (I)
punto+no terminal

Ejemplo

Siguiendo la gramática habitual:

I = {E → ·E + T}

Cierre (I) = {E → ·E + T, ~~E → ·E + T~~, E → ·T, T → ·T * F, T → ·F, F → ·(E), F → ·id}

No se incluyen los repetidos.

El cierre de un estado son todas las posibilidades del lenguaje que se pueden aplicar desde una determinada regla.

Función Goto

Goto (I, x) x ϵ N U T

Goto (I, x) = Cierre del conjunto formado por todos los ítems de la forma: A → αx·β tales que A → αx·β ϵ I

Ejemplo

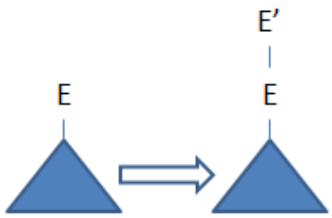
I = { E → ·E + T, E →·T}

Goto (I, +) = Cierre (E →E+·T) = {E→E+·T, T → ·T * F, T → ·F, F → ·(E), F → ·id}

Gramática aumentada

G(N, T, P, S) →Gramática aumentada → G'(N U {S'}, T, P U {S'→ S}, S')

Es un formulismo que utilizamos al crear un LR. Esta gramática aumentada nos asegura que hemos llegado al final cuando usamos la reducción por la regla E' → E



Podría haber otras producciones con E, pero solo una con E'.

Autómata reconocedor de prefijos viables

Colección canónica

Conjunto de ítem inicial I_0=Cierre{S' → ·S}

Colección canónica CC = {I_0} J=1 → contador

Repetir

∀I_i ϵ CC ∀x ←N U T

I_j = Goto(I_i, x)

Si I_j ϵ CC, I_j ≠ Ø → CC = CC U {I_j}, j = j+1

Hasta que CC no cambie.

Ejemplo

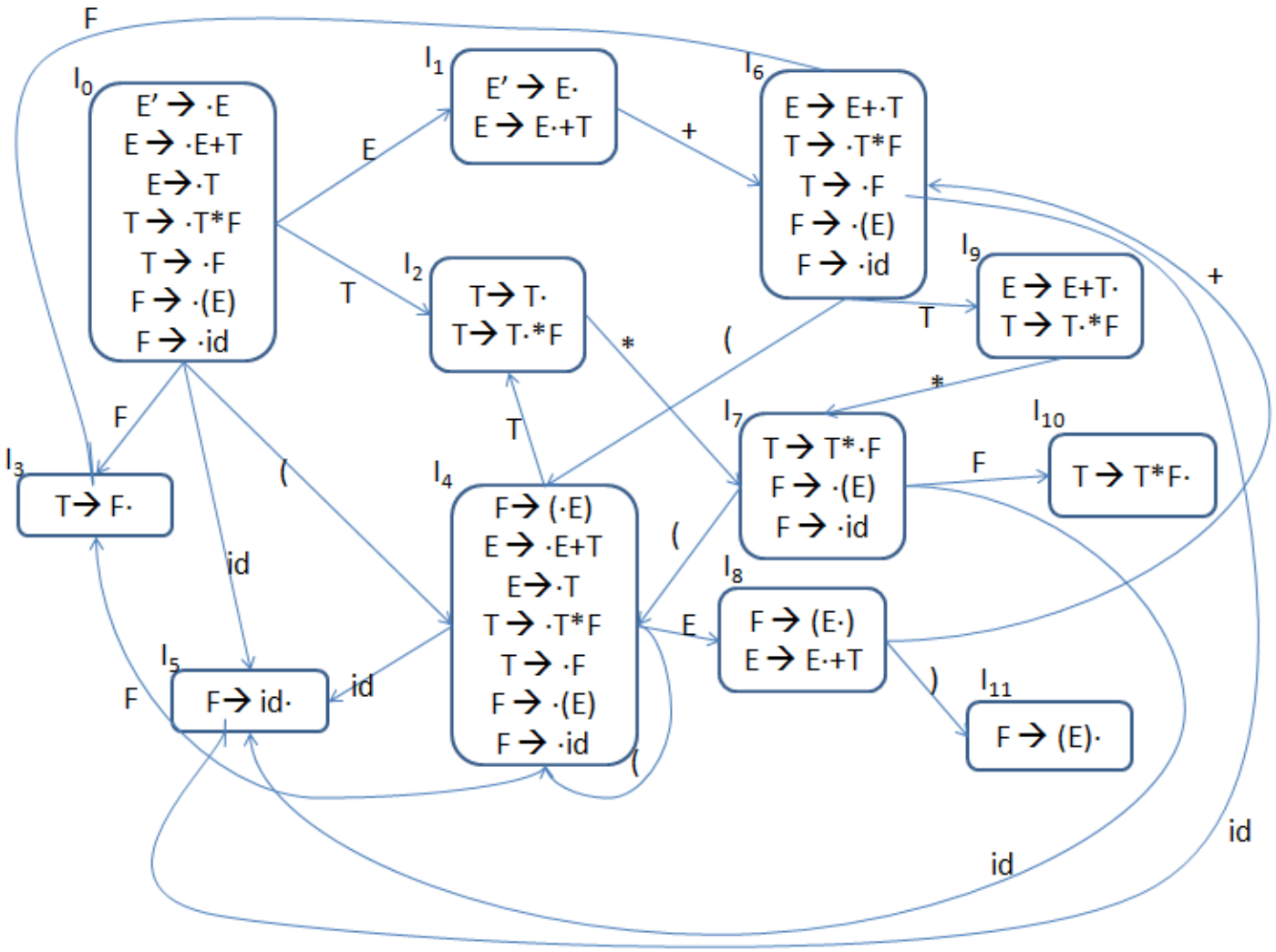
I_0 = Cierre {E' → ·E} = {E'→·E, E → ·E + T, E → ·T, T → ·T * F, T → ·F, F → ·(E), F → ·id}

CC={I_0} j = 1

I_0 = Goto (I_0, E) = Cierre {E'→E·, E → E ·+ T} = {E'→E·, E → E ·+ T} = I_1
I_0 = Goto (I_0, T) = Cierre { T → T ·* F, E → T·} = { T → T ·* F, E → T·} = I_2
I_0 = Goto (I_0, F) = Cierre {T → F·} = {T → F·} = I_3
I_0 = Goto (I_0, +) = Cierre {Ø} = {Ø}
I_0 = Goto (I_0, *) = Cierre {Ø} = {Ø}
I_0 = Goto (I_0, '(') = Cierre { F → (·E)} = { F → (·E), E → ·E + T, E → ·T, T → ·T * F, T → ·F, F → ·(E), F → ·id} = I_4
I_0 = Goto (I_0, ')') = Cierre {Ø} = {Ø}
I_0 = Goto (I_0, id) = Cierre {F → id·} = {F → id·} = I_5

CC = { I_0, I_1, I_2, I_3, I_4, I_5,…} y continuaría, con esto solo terminamos el bucle en el estado I_0 habría que hacer nuevos bucles para cada nuevo estado.

Expresado de forma gráfica cada I_j sería un estado del autómata.



Ejercicio

Crear un autómata reconocedor de prefijos viables.

A → BC
B →dB / b
C → dB / C / dC

First

First (α) = {α ϵ T / ∃α →* aβ} Si α →* λ, λ ϵ First(α)

Es el conjunto de símbolos terminales que encabezan cadenas que se producen desde cualquier símbolo.

Ejemplo
Dada la gramática habitual:

E → E + T
E → T
T → T * F
T → F
F → (E)
F → id

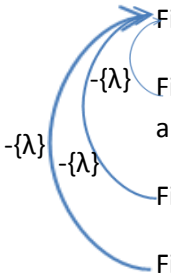
First(+) = {+}
First(*) = {*}
First(id) = {id}
} El First(α) donde αϵT es siempre y solo α.

First(E) = {(, id}
First(T) = {(, id}
First(F) = {(, id}

Aquí miramos siempre el símbolo N a la izquierda de la regla.

Cuestión particular en First(S)

S → ABC
A →λ
B → A
C → A
C → +



First (S) = {+, λ}

First (A) = {λ} → Esto quiere decir que las cadenas de S que empezaban por A y A se anula hacen que las cadenas de S empiecen por B

First(B) = {λ} → Como B también se anula, las cadenas de S empiezan por C

First(C) = {+, λ}

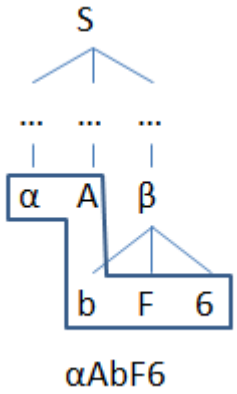
Follow

Conjunto de símbolos terminales que pueden ir a continuación del no terminal en alguna fórmula válida del lenguaje:

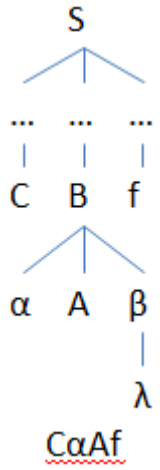
Follow (A) = {α ϵ T / S →* αAaβ} A ϵ N

Aplicaremos una serie de reglas:

- $ ϵ Follow(S) en cualquier caso.
- Si ∃B → αAβ → First(β) ϵ Follow(A)



- Si ∃B → αA ó ∃B → αAβ ∧ λ ϵ First(β) => Follow(B) ⊆ Follow(A)



Aplicando estas tres reglas podemos calcular el follow de cualquier no terminal. Solo funciona con no terminales. En este caso siempre se buscan las reglas donde el símbolo N aparezca a la derecha de la regla.

Ejemplo

Dada la gramática de la página 33:

Follow (E) = {$, +,)} Si a la derecha del no terminal aparece un terminal este se añade directamente. $ se añade por ser axioma siempre.

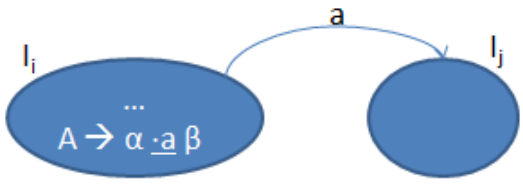
Follow (T) = {$, +,), *} Se añade el Follow(E) ya que T puede no tener nada a su derecha y entonces se añade al Follow(T) el Follow del padre.

Follow (F) = {$, +,), *S} Se añade el Follow(T) por el mismo motivo.

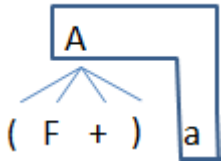
Tabla de ACCIÓN

El estado S_i se obtiene de I_i teniendo en cuenta que:

- Si A→ α ·a β ϵ I_i ∧ Goto (I_i, a) = I_j → Acción(i, a) = desplazar e ir a j
  punto + T



- Si A → α· ϵ I_i → Acción(i, a) = reducir (A → α)
  ∀a ϵ Follow(A)

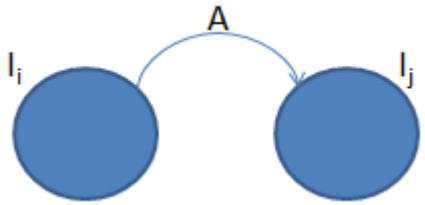


Por tanto, si a no pertenece al Follow(A) entonces sería un error.

- Si S' → S· ϵ I_i => Acción(I, $) = ACEPTAR

Tablas de GOTO

Goto(I_i, A) = I_j → GOTO(i, A) = j



S' → ·S ϵ Estado inicial I_0

Al construir la tabla de decisión en cada estado pueden pasar varias cosas:

- Aparece una sola acción → Bien. Se coloca sin ningún problema.
- No aparece nada →Bien. Indica que es un lugar vacio, si en la pila se llega a ese lugar se detectará un error.
- Aparecen dos o más acciones → Mal. Conflicto. Significa que la gramática es ambigua y tenemos un problema.

Ejemplo

Usando el autómata de prefijos viables de la página 32.

Tenemos tantos estados como nodos en el autómata.

	ACCIÓN						GOTO		
estado	id	+	*	(	)	$	E	T	F
S_0	D5			D4			1	2	3
S_1		D6				ACEPTAR			
S_2		R2	D7		R2	R2			
S_3		R4	R4		R4	R4			
S_4	D5			D4			8	2	3
S_5		R6	R6		R6	R6			
S_6	D5			D4				9	3
S_7	D5			D4					10
S_8		D6			D11				
S_9		R1	D7		R1	R1			
S_{10}		R3	R3		R3	R3			
S_{11}		R5	R5		R5	R5			

Para cada estado uno por uno vamos utilizando las reglas anteriormente descritas sobre el autómata. Cada transición en el autómata de un estado a otro es un desplazamiento.

Una transición con símbolo no terminal es un nuevo estado en GOTO

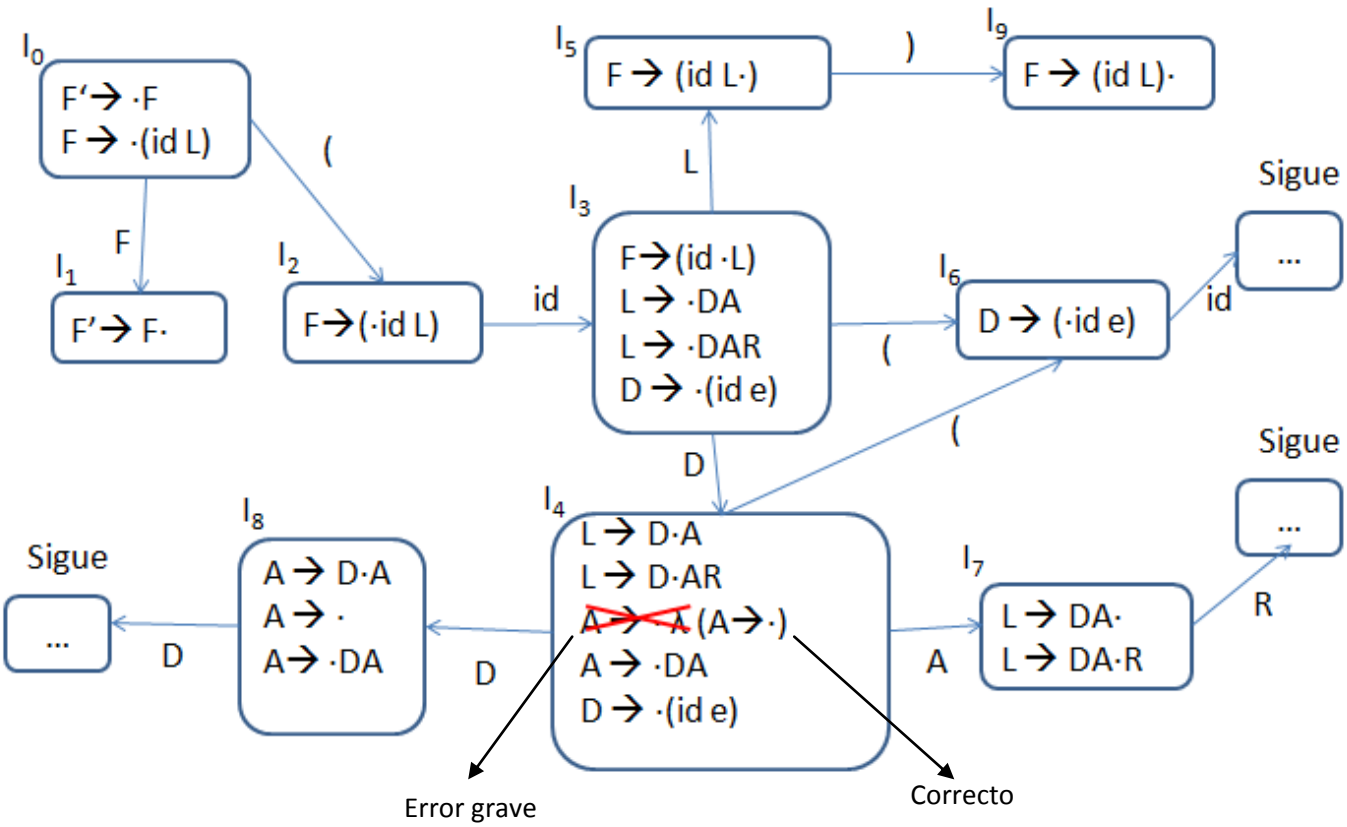
Cuando encontramos un ítem con punto al final reducimos por la regla donde esté ese ítem en los Follow del símbolo a la izquierda de la regla. El número junto a r indica el número de regla.

Ejercicio de examen

Dada la gramática:

F → (id L)
L → DA / DAR
A → λ / DA
D → (id e)
R → rest A

Primero aumentamos la gramática:
F' → F



Ejercicio

S → SA / b

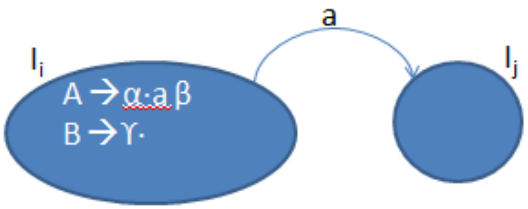
Construir autómata y tabla de decisión.

A → Aa / Ab / B

B → dBe / e

Conflictos LR

Reducción/Desplazamiento

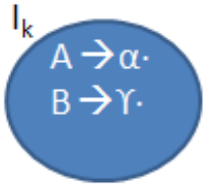


Acción[I_i, a] = desplazar e ir I_j
Acción[I_i, Follow(B)] = Reducir B →γ

Pero, ¿qué sucede si a ϵ Follow(B)?

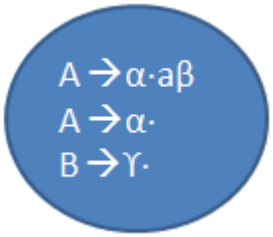
Tendremos dos acciones distintas en un mismo lugar. Eso no es posible, es un conflicto de tipo Reducción/Desplazamiento.

Reducción/Reducción



Follow(A) ∩ Follow(B) ≠ ∅ → Conflicto Reducción/Reducción.

Hay que tener cuidado al aplicar las condiciones:



Si decimos que:

{a} ∩ Follow(A) ∩ Follow(B) = ∅ → No hay Conflicto.

Puede parecer correcto, pero **NO**, con esta condición solo vemos si existen conflictos de tres posibilidades, pero no si hay de dos (que es lo más común). Para ello habría que desgranar esa condición en tres:

- {a} ∉ Follow(A)
- {a} ∉ Follow(B)
- Follow(A) ∩ Follow(B) ≠ ∅

} No hay conflictos

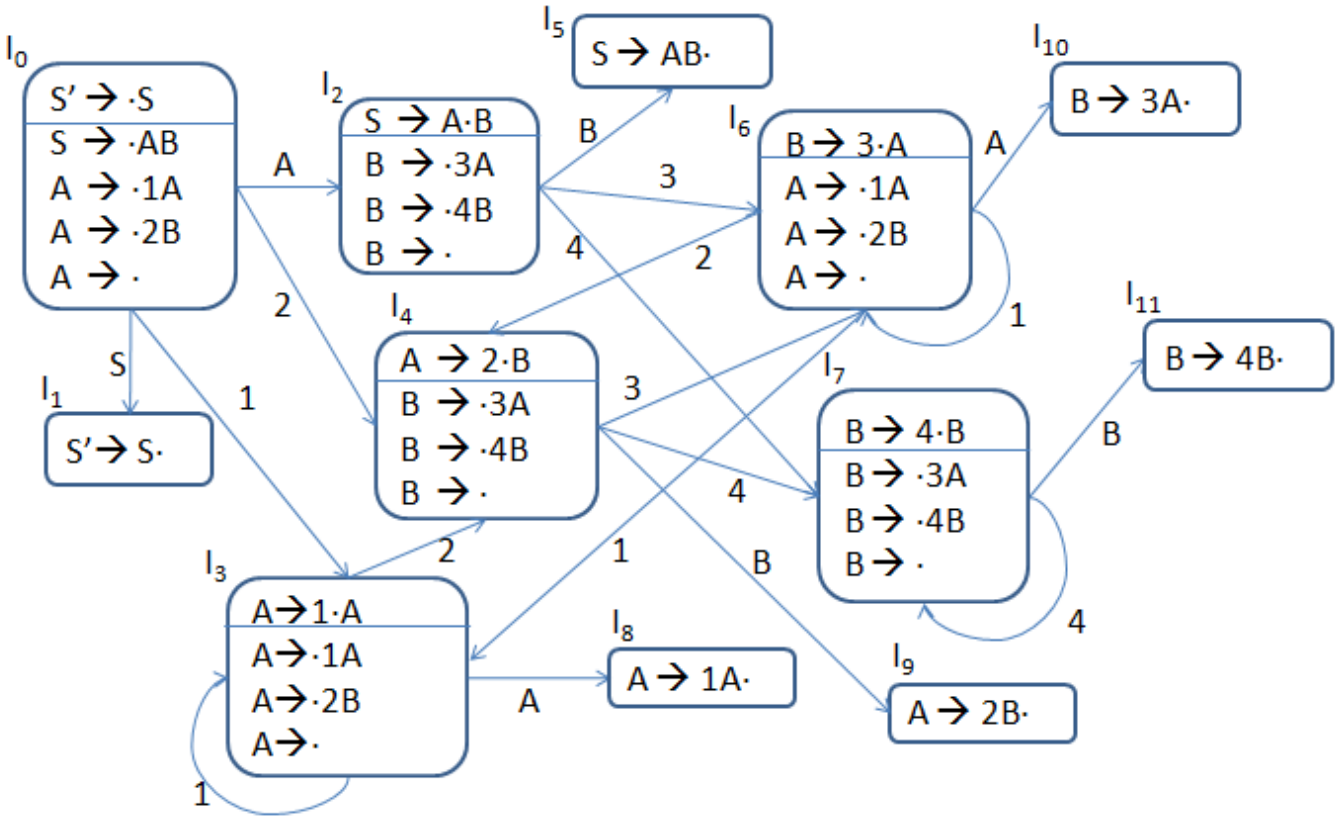
Se recomienda encontrar los conflictos antes de hacer la tabla de decisión, para ahorrar trabajo y coste, y verificar las condiciones tras el autómata de prefijos viables.

Ejercicio de examen

Dada la siguiente gramática:

S → AB
A → 1A / 2B / λ
B → 3A / 4B / λ

Construimos la gramática aumentada:
S' → ·S



First (S) = {1, 2, 3, 4, λ} Follow (S) = {$}

First (A) = {1, 2, λ} Follow (A) = {3, 4, $}

First (B) = {3, 4, λ} Follow (B) = {3, 4, $}

Susceptibles de problemas

I_0 → Red/desp I_3 → Red/Desp I_6 → Red/Desp

I_2 → Red/desp I_4 → Red/Desp I_7 → Red/Desp

De los cuales darán error → {I_2, I_3, I_4}

En el análisis LR no existen conflictos de desplazamiento/desplazamiento.

Análisis sintáctico descendente

Condición necesaria

Para que una gramática pueda analizarse por método descendente es necesario que no presente recursividad por la izquierda (explicado en la página 6) y que esté factorizada por la izquierda.

Factorización por la izquierda

Si para un mismo no terminal existen dos reglas que empiezan por el mismo símbolo esa gramática no está factorizada por la izquierda:

S → AB
A → aC / aD ← Dos producciones empiezan por 'a' NO está factorizada.

Si factorizamos:
S → AB
A → aA' Similar al concepto de sacar factor común.
A' → C / D

Análisis sintáctico descendente - Predictivo recursivo

A la hora de diseñar los procesos de cada no terminal A:

1. Decidir que regla de A aplicar
   a. Si siguiente token ϵ First(α) => aplicar A → α
   b. Si siguiente token ∉ First(α / A → α ϵ G) => aplicar A → λ (Si existe)
   c. Si siguiente token ϵ First(α) ∧ ϵ First(B) A → α, A → B ϵ G => ERROR
      (Debe ser determinista)
2. Para cada símbolo de la derecha
   a. No terminal → Llamada a su proceso.
   b. Terminal → Debe coincidir con el siguiente token.

Ejemplo

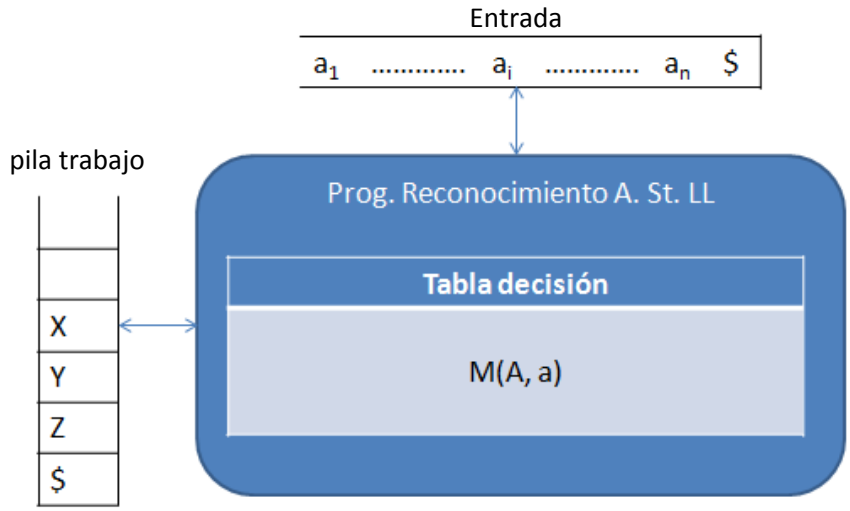
E → E+T / T
T → T*F / F
F →(E)/id

Primero eliminamos recursividad por la izquierda:
E → TE'
E' → +TE' / λ
T → FT'
T' → *FT' / λ
F → (E) / id Sin problemas de factorización.

Programa Analizador sintáctico descendente PR

```
leer siguiente token (a);
      E;
End;
Proc E;
      T;
      E';
End;
Proc E';
      If a = First(+TE') then
            Begin
                  leer siguiente token(a);
                  T;
                  E';
            End;
      Else Ø; //Aplicación de E' → λ
End;
//El procedimiento T' sería similar con a=*
Proc F;
      If a = ( then
            leer siguiente token(a);
            E;
            leer siguiente token(a);
                  if a = ) leer siguiente token(a);
                  Else ERROR;
                  End;
      Else if a=id then leer siguiente token(a);
      Else ERROR;
      End;
End;
```

Análisis sintáctico descendente – LL



Acciones

Si CimaPila = siguiente token = $ → Fin con éxito.

Si CimaPila = siguiente token ≠ $ → Sacar CimaPila, leer siguiente token

Si CimaPila ϵ N → Consultar M[CimaPila, siguiente token] = | Ø
CimaPila → XYZ Sustituir CimaPila por derecha de la regla.

Construcción tablas decisión LL

∀A → α ϵ G

- Si t ϵ First(α) → M(A, t) = A →α (t ϵT)
- Si λ ϵ First(α) → M(A, t) = A → α ∀t ϵ Follow(A)

Ejemplo

First (E) = {(, id}
First (E') = {+, λ}
First (T) = {(, id}
First (T') = {*, λ}
First (F) = {(, id}

Follow (E) = {), $}
Follow (E') = {), $}
Follow (T) = {+,), $}
Follow (T') = {+,), $}
Follow (F) = {*, +,), $}

E → TE'
First (TE') = {(,id}
Se hace el First de cada producción de las reglas y esta regla se sitúa en la table bajo el símbolo que sea su First.

	+	*	(	)	Id	$
E			E → TE'		E → TE'	
E'	E' → +TE'			E' →λ		E' →λ
T			T → FT'		T →FT'	
T'	T' → λ	T' → *FT'		T' →λ		T' →λ
F			F → (E)		F → id	

Ejercicio

F →(id A)
A → P O R
O → P O / null
R →rest O / λ
P → (id O)

Crear el analizador sintáctico LL.

Condición LL

- Si A → α_1, A → α_2 ϵ G => First(α_1) ∩ First(α_2) = Ø Es decir, que no tengan nada en común.
- Si A→ α_1, A → α_2 ϵ G y λ ϵ First(α_1) => Follow(A) ∩ First(α_2) = Ø
  Pertenece a una de las reglas.

Si existen n producciones de A debe examinarse la condición con todas las parejas de producciones posibles.

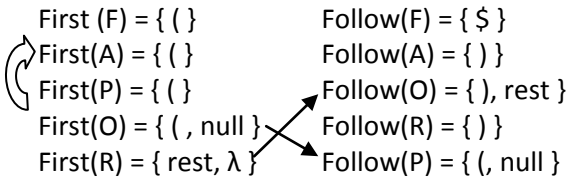
A → α_1 / α_2 / … / α_n => First(α_1) ∩ First(α_2) = Ø
First(α_1) ∩ First(α_3) = Ø
…
First(α_1) ∩ First(α_n) = Ø
First(α_2) ∩ First(α_3) = Ø
…

Si $\lambda \in First(\alpha_1)$(o a una de ellas, solo una) debería examinarse:

Follow (A) ∩ $First(\alpha_2)$ = ∅
...
Follow (A) ∩ $First(\alpha_n)$ = ∅

Ejemplo

F → (id A)
A → POR
O → PO / null
R → rest O / λ
P → (id O)



Aplicamos la condición LL.

Las únicas susceptibles serían O y R por tener más de una producción, entonces:

¿ First(PO) ∩ First(null) = ∅ ?
{ (} ∩ {null} = ∅ → La O no da problemas.

¿ First(rest O) ∩ First(λ) = ∅ ?
{ rest } ∩ {λ} = ∅ → La primera condición es correcta pero First(λ) = λ así que debemos examinar la segunda condición.

¿ First(rest O) ∩ Follow(R) = ∅ ?
{ rest } ∩ {) } = ∅ → La R no da problemas

La pregunta es, si incluimos la regla P → λ, ¿Qué cambia?

En este caso First(P) = { (, λ } entonces al poder anularse P → First(A) = { (, null }

y al examinar la condición:

¿First(PO) ∩ First(null) = ∅ ?
{ (, null} ∩ {null} ≠ ∅

y, además habría que preguntarse:

¿First((id O)) ∩ Follow(P) = ∅ ?
{ (}∩ { (, null } ≠ ∅

No cumpliéndose entonces la condición LL.

Ejemplo de examen

BLOQUE → (block SENT REST-BLOCK)
REST-BLOCK → SENT REST-BLOCK / return / λ
SENT → SENT-ASIG / EXPRESION / SENT-CONTROL

First(SENT-ASIG) = First(EXPRESION) = { id }
SENT-ASIG →* id op-asig id
EXPRESION →* id op-arit id
First(SENT-CONTROL) = { If, while, for }

	(	)	block	return	id	if	while	for	$
BLOQUE	BLOQUE → (…								
REST-BLOCK		REST-BLOCK → λ		REST-BLOCK →return	REST-BLOCK →SENT…	REST-BLOCK →SENT…	REST-BLOCK →SENT…	REST-BLOCK →SENT…	
SENT				SENT→SENT-ASIG SENT → EXPRESION	SENT→ SENT-CONTROL	SENT → SENT-CONTROL	SENT → SENT-CONTROL		
SENT-ASIG									
EXPRESION									
SENT-CONTROL									

En (SENT, id) hay dos reglas en la misma casilla, por tanto, esta gramática no cumple la condición LL y por ello no permitiría el análisis LL.

First(BLOQUE) = { (}
First(REST-BLOCK) = { return, λ, id, if, while, for }
First(SENT) = { id, if, while, for }

Follow(BLOQUE) = { $ }
Follow(REST-BLOCK) = {) }
Follow(SENT) = { return, id, if, while, for }

Ejercicio

S →procedure division . [DECLARACIONES] {CUERPO}
DECLARACIONES → declaratives . {STRUCT use PARRAFO} end declaratives
CUERPO → id section . {PARRAFO}
STRUCT →number . SENTENCE
SENTENCE → goto id / evaluate id

Esta gramática está en formato extendido, donde lo que está entre [](corchetes) significa que puede aparecer 1 o ninguna vez y lo que está entre {}(llaves) 0 ó n veces. Las palabras en mayúsculas son No Terminales y en minúscula Terminales.
Se trata de hacer el algoritmo predictivo-recursivo para esta gramática.

Análisis sintáctico descendente por tablas

Dada la gramática de la página 42 y la palabra:

(id (id null) null)

Creada a partir de esa gramática el análisis por tablas es el siguiente:

PILA	Cadena	Regla
$ F	~~(~~id (id null) null)	F → (id A)
$) A id~~(~~	~~id~~ (id null) null)	
$) A ~~id~~	(id null) null)	
$) A	(id null) null)	A → POR
$) R O P	~~(~~id null) null)	P → (id O)
$) R O) O id~~(~~	~~id~~ null) null)	
$) R O) O ~~id~~	null) null)	
$) R O) O	~~nu~~ll) null)	O → null
$) R O) ~~nu~~ll	~~)~~ null)	
$) R O~~)~~	null)	
$) R O	~~nu~~ll)	O → null
$) R ~~nu~~ll	)	
$) R	~~)~~	R → λ
$~~)~~	-	

La producción de cada regla se coloca en orden inverso en la pila { (id A) →)A id(} para que en la cabeza de la pila (que crece hacia la derecha) esté el primer símbolo de la producción.

Si el símbolo de la pila coincide con el token dado, vamos por el buen camino, el análisis está siendo correcto. Si al contrario no coinciden, es un error de la cadena.

Ejemplo de análisis predictivo-recursivo

```
Comprobar_token(T:Token)
BEGIN
      If t=sgte_token;
      Then leer_sgte_token(sgte_token);
      Else error;
END

PROGRAM Ejemplo Desc_Rec;
      Leer_sgte_token(Sgte_token);
      BLOQUE;
END
Procedure BLOQUE;
BEGIN
      Comprobar_token('(');
      Comprobar_token('Block');
      SENT;
      REST-BLOCK;
      Comprobar_token(')');
END
Procedure SENT;
      If(sgte_token ∈ {If, While, For})Then
            S-CONTROL;
```

```
      Else if (sgte_token == id) Then
            BEGIN                        //Aquí habría un conflicto
porque pueden ser dos reglas.
                  Comprobar_token(id);
                  If(sgte_token == op-asig) Then
                        BEGIN
                              Comprobar_token(op_asig);
                              Comprobar_token(id);
                        END
                  Else
                        BEGIN
                              Comprobar_token(op_arit);
                              Comprobar_token(id);
                        END
            END
END
Procedure REST-BLOCK;
BEGIN
      If(sgte_token ϵ {id, if, while, for) Then
            BEGIN
                  SENT;
                  REST-BLOCK;
            END
      Else if (sgte_token == result) Then
            Comprobar_token(return);
      Else //if sgte_token ϵ Follow(REST-BLOCK)
END
```

Una solución para eliminar el conflicto sería cambiar la gramática por una equivalente:

SENT → S-CONTROL / id S-AUX

S-AUX → Resto_asig / Resto_expr
op_asig id / op_arit id

Así la gramática volvería a ser LL

Análisis semántico

En el análisis semántico se utilizará una gramática de contexto libre (tipo 2) pero con un añadido denominado gramática de atributos. Estos atributos que añadiremos a los no terminales nos darán información sobre su contenido semántico.

Estos atributos se otorgan a los símbolos mediante las **reglas semánticas**:

Reglas semánticas →
- Calcular el valor de los atributos.
- Hacer comprobaciones sobre los valores.
- Acciones laterales →
  - Guardar información en la TS.
  - Emitir mensajes de error.
  - Generar código.

En análisis semántico genera, al final, un formalismo denominado Traducción dirigida por la sintaxis.

Traducción dirigida por la sintaxis

Se presenta con dos notaciones diferentes:

- Definición dirigida por la sintaxis (DDS) → Notación de alto nivel con menos detalles.
- Esquema de traducción (EDT) → Notación de bajo nivel con más detalles.

Lo que estamos trabajando es el diseño del analizador semántico, lo que quiere decir que si trabajamos con DDS nos tendremos que preocupar de menos cosas pero será cosas que tendremos que definir más tarde.

Todo lo contrario es trabajar con EDT, durante el diseño da más trabajo pues te obliga a trabajar con más detalles pero allana el camino para más tarde.

Atributos

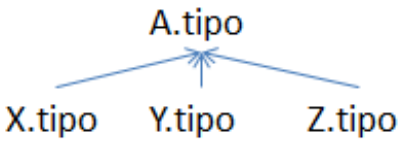
Se aplican sobre los no terminales:

A → XYZ donde puedo definir un atributo tipo (por ejemplo):

A. tipo = f(x.tipo, y.tipo, z.tipo)

Los atributos pueden ser de dos tipos:

- **Sintetizados**: Se calculan, exclusivamente, a partir de los atributos del lado derecho de la regla. A partir de los hijos.



Como es el caso del ejemplo.

- **Heredados**: Cualquier atributo no sintetizado. Se calculan a partir de cualquier tipo de atributo, no exclusivamente de los hijos sino también de los hermanos y el padre.

X. Tipo = f(Y.tipo) X.Posición = A.posición

Ejemplo

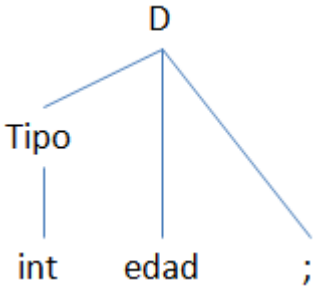
Dada la frase:

int edad;

con la gramática:

D → Tipo id;
Tipo → int/float

Árbol sintáctico:



Hasta ahora solo nos fijábamos en si la estructura es correcta, ahora debemos definir que el tipo es entero(int).

Tenemos que unir de alguna manera el tipo ese id, la única manera es con un atributo que recorra las reglas.

D → Tipo id; id.tipo = tipo.tipo (Heredado)

Tipo → int Tipo.tipo = int

Tipo → float Tipo.tipo = int

Veámoslo con un ejemplo:

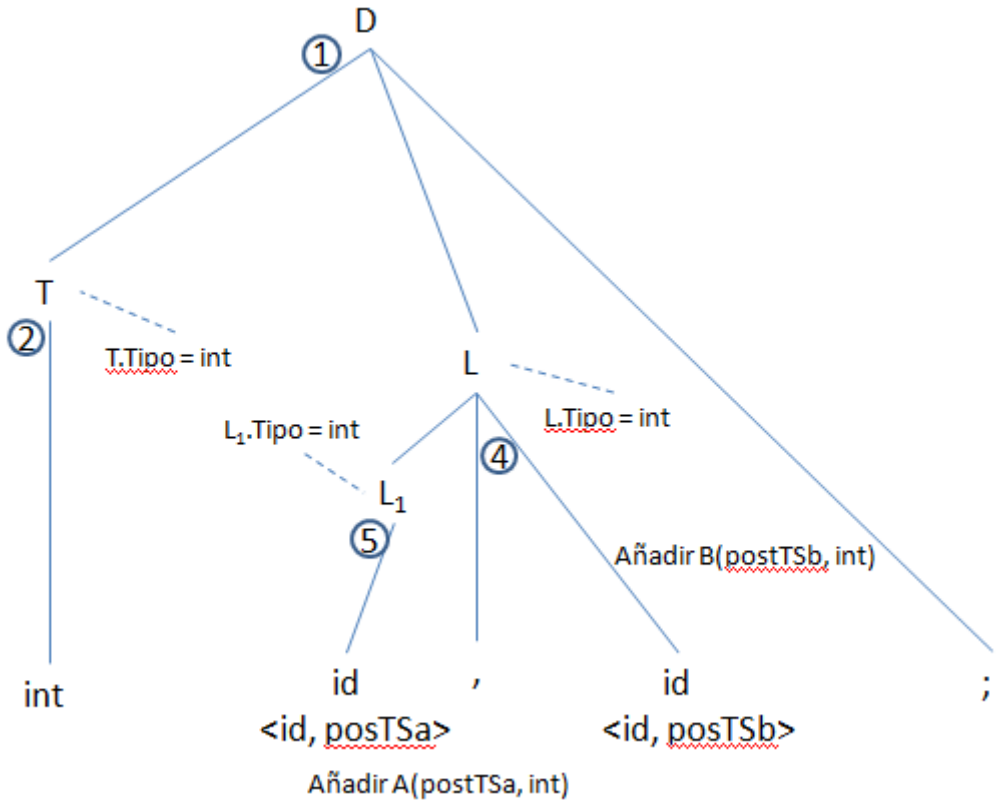
D → T L ;	{L.tipo = T.tipo}
T → int	{T.tipo = int}
T → float	{T.tipo = float}
L → L_1, id	{L_1.tipo = L.tipo; Añade tipo TS(id.posTS, L.tipo)}
L → id	{Añade tipo B(id.posB, L.tipo)}

Como se puede observar en la regla 4, diferenciamos (mediante subíndices) los símbolos no terminales iguales en una misma regla:

L → L, id

son el mismo símbolo
pero diferentes nodos.

L → L_1, id Dada la palabra *int a, b;*



Nos faltaría saber muchas cosas aún, la primera de ellas: ¿Qué orden seguimos? En qué momento se hace cada cosa difiere mucho según se utilice DDS o EDT.

Definiciones con atributos por la izquierda

∀ atributo heredado de X_j, A → $X_1 \ldots X_j \ldots X_n$

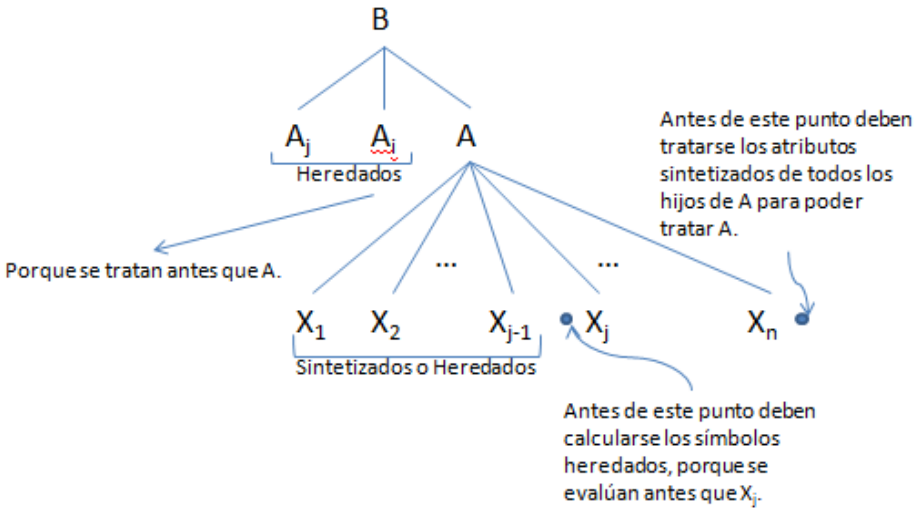
Su cálculo depende solo de:

- Los atributos (Sintetizados o heredados) de $X_1 \ldots X_{j-1}$
- Los atributos heredados de A.

La idea que se persigue es que, cuando llegue el momento de expandir un nodo ya tenemos toda la información relevante sobre él.

Como consecuencia, deben cumplirse las siguientes condiciones cuando hay atributos sintetizados y heredados:

- Un atributo heredado para un símbolo del lado derecho de la regla debe calcularse antes de que se trate dicho símbolo.
- Un atributo sintetizado para el no terminal de la izquierda "sólo" puede calcularse una vez tratados todos sus hijos.



Formas de representación

D → T L ; L.tipo = T.tipo
T → int T.tipo = int
T → float T.tipo = float
L → L_1, id L_1.tipo = L.tipo; Añade tipo TS(id.PosTS, L.tipo)
L → id Añade tipoTS (id.PosTS, L.tipo)

DDS → Tenemos reglas y acciones semánticas pero no nos preocupamos de cuando se implementa cada acción semántica.

En EDT sí debemos preocuparnos de este detalle. Tenemos que utilizar la estrategia de atributos por la izquierda, y resulta:

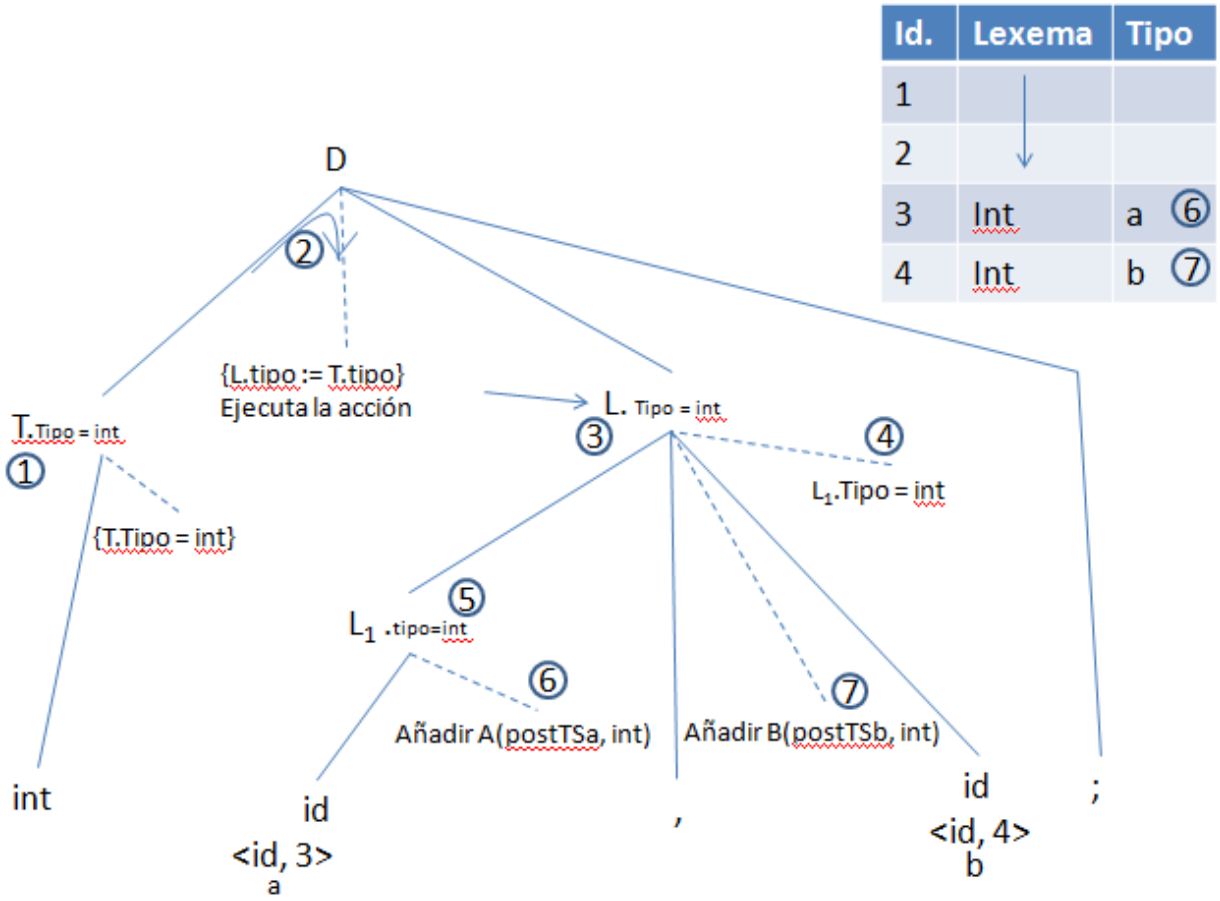
D → T{ L.tipo = T.tipo} L ;
T → int {T.tipo = int}
T → float {T.tipo = float}
L → { L_1.tipo = L.tipo } L_1, id {Añade tipo TS(id.PosTS, L.tipo)}
L → id {Añade tipoTS (id.PosTS, L.tipo)}

Las acciones laterales siempre se evalúan al final.

En EDT, entonces, cada acción semántica se coloca en la gramática exactamente cuando se debe hacer uso de ellas.

Cómo funciona gráficamente:

Dada la palabra int a, b;



Aunque en este caso nos hemos permitido hacer un análisis descendente aunque no sería del todo posible si la frase fuese más larga.

Comprobaciones semánticas

Sistemas de tipos

Orientado a la comprobación de tipos. No es más que un conjunto de reglas cuyo propósito es asignar una expresión de tipo a las distintas estructuras del texto fuente, en el caso que nos ocupa a las estructuras del lenguaje de programación.

El comprobador de tipos (subconjunto muy grande del analizador semántico, incluso podemos generalizar que es el analizador por completo) es la implementación de sistemas de tipos.

Expresión de tipos

Denota el tipo de una construcción del texto fuente. Existen dos tipos.

Tipos básicos

Todos aquellos tipos definidos dentro del propio lenguaje. Aunque existen tres tipos básicos especiales:

- Tipo_OK
- Tipo_error
- Tipo_vacio

Estos tipos básicos especiales nos ayudan a conocer la semántica de ciertas sentencias, que no son identificadores ni datos.

Tipos construidos

Se basan en la aplicación de un constructor a uno o más tipos básicos o construidos.

Constructores de tipos

Veamos los casos más comunes:

Vectores

Si T es expresión de tipo → array(I, T) es expresión de tipo (donde I son los índices).

Ejemplo

char V[10] → array(1..10, char) se asigna a V.

Producto cartesiano

Si T_1, T_2 son expresiones de tipo → T_1X T_2 es expresión de tipo.

Este constructor nace más como una necesidad del compilador que como expresión del lenguaje. Luego veremos por qué.

Registro

Si T_1,…,T_n son expresiones de tipo → record (T_1xT_2x…xT_n) es expresión de tipo

Ejemplo

```
Struct R
 { int a;          → record(int x array(1..5, real)) → R
   float b[5]                                ó 0..4 esto dependen del lenguaje.
 }
```

Punteros

Si T es expresión de tipo → pointer(T) es expresión de tipo.

Funciones

int ejemplo (float, char) → float x char --> int es expresión de tipo que se asigna a ejemplo.

Ejemplo

P → D ; S **Declaración ; sentencias**

D → D_1 ; D_2 / id:T

T → char / integer / boolean / ↑T_1 / array[num] of T_1 **básicos**, **puntero**, **array**, **construidos**

S → S_1 ; S_2 / id:=E / if E then S_1 **asig**, **if-then**, **Sentencias**

E → car / num / true / false / id / E_1 mod E_2 / E_1[E_2] / E_1↑ / E_1 and E_2 / E_1 op.rel E_2 **operaciones**: **modulo**, **puntero**, **lógica**, **general**

Recordemos que los subíndices sirven para diferenciar los no terminales con mismo nombre en una regla.

Comprobación de tipos

D → id:T {añade tipo (id.entrada, T.tipo)}

T → char {T.tipo = char}

T → integer {T.tipo = integer}

$T \rightarrow$ boolean {T.tipo = boolean}

$T \rightarrow \uparrow T_1$ {T.tipo = pointer(T_1.tipo)} **Como se ve, tipo construido.**

$T \rightarrow$ array[num] of T_1 {T.tipo = array(1..num.valor, T_1.tipo)}

$S \rightarrow S_1 ; S_2$ {S.tipo := (If S_1.tipo = tipo_OK and S_2.tipo = tipo_OK then tipo_OK Else tipo_error)}

$S \rightarrow$ id := E {id.tipo = BuscaTipoTS(id.entrada) ; S.tipo := (If id.tipo = E.tipo Then tipo_OK Else tipo_error)}

$S \rightarrow$ If E then S_1 {S.tipo := (If E.tipo = Boolean and S_1.tipo = tipo_OK Then tipo_OK Else tipo_error)}

$E \rightarrow$ car {E.tipo = char}

$E \rightarrow$ num {E.tipo = integer} **(Solo hay expresiones de tipo entero en el lenguaje)**

$E \rightarrow$ true {E.tipo = boolean}

$E \rightarrow$ false {E.tipo = boolean}

$E \rightarrow$ id {E.tipo = buscaTipoTS(id.entrada)}

$E \rightarrow E_1$ mod E_2 {E.tipo := (If E_1.tipo = integer and E_2.tipo = integer Then integer Else tipo_error)}

$E \rightarrow E_1[E_2]$ {E.tipo := (If E_2.tipo = integer and E_1.tipo = array(i,t) Then t Else tipo_error)} **(Este es un caso muy sencillo, en un caso real habría que hacer más operaciones)**

$E \rightarrow E_1\uparrow$ {E.tipo := (If E_1.tipo = pointer(t) Then t Else tipo_error)}

$E \rightarrow E_1$ and E_2 {E.tipo := (If E_1.tipo = boolean and E_2.tipo = boolean Then boolean Else tipo_error)}

$E \rightarrow E_1$ op.rel E_2 {E.tipo := (If E_1.tipo = integer and E_2.tipo = integer Then boolean Else tipo_error)}

Podemos añadir la regla para sumar:

$E \rightarrow E_1 + E_2$ {E.tipo := (If E_1.tipo = integer and E_2.tipo = integer Then integer Else tipo_error)}

¿Qué sucedería si hubiese más tipos de números?

Si existe int/float

$E \rightarrow E_1 + E_2$ {E.tipo := (If E_1.tipo = int and E_2.tipo = int Then int
Else if E_1.tipo = float and E_2.tipo = float Then float
Else tipo_error)}

Este es un ejemplo con un lenguaje sin conversión automático de tipos.

Con conversión automática de tipos sería:

$E \rightarrow E_1 + E_2$ {E.tipo := (If E_1.tipo = int and E_2.tipo = int Then int
If E_1.tipo = int and E_2.tipo = float Then float
If E_1.tipo = float and E_2.tipo = int Then float
If E_1.tipo = float and E_2.tipo = float Then float
Else tipo_error)}

Ejercicio

C : char;
i : integer;
i : c mod i mod 3

Ejercicio resuelto

P → DP / SP / λ
D →function id (L) : T ; begin P end
D → T id;
L → T : id / L ; L
T → integer / real
S → id := E;
E → id (A) / id
A → E / A, A

Lenguaje paso de parámetros por valor. Sin conversión automática de tipos.

Construir EDT, detallando accesos a TS incluyendo construcción y destrucción y teniendo en cuenta que el analizador léxico no introduce nada en la tabla de símbolos.

EDT

- Comprobaciones de tipo.
- Comprobaciones de que las sentencias son correctas.
- Manejar TS.
- Insertar id en TS, y la información asociada.
  - ¿Dónde estará cada id en tipo de ejecución?

Lo primero que se debe hacer es iniciar todas las pilas y variables, esto suele hacerse en la primera regla, antes de la primera producción, pero en este caso, la primera regla no siempre se utiliza solo al principio. Por ello hay que colocar las sentencias iniciales en otro lugar. Hay dos soluciones:

1. Inicializar todo antes de comenzar a trabajar. Dentro del código del analizador semántico pero no dentro de la gramática.
2. Aumentar la gramática con P'→P y inicializar todo en esta regla, antes de expandir por P.

Utilizaremos la segunda opción para que así esté dentro del esquema de traducción.

Entonces:

En este ejercicio se permiten funciones dentro de funciones. Por tanto se crearán tablas de símbolos anidadas para cada una de ellas. Para poder manejar esto habrá que crear una pila de tablas de símbolos.

Que se construyen y se destruyen según se usen o se dejen de usaren el código.

También debe crearse una pila que contabilice el desplazamiento de cada elemento en una TS.

P' → { PilaTS := nuevaPila()
PilaDesp := nuevaPila()
push (PilaTS, crearTS())
push (PilaDesp, 0) **} P**

P → DP /SP / λ (No vamos a colocar ninguna acción en este punto, aunque podríamos verificar que no se comenten errores y cosas así).

D → function id { If buscarTS (cima (PilaTS), id.lexema) ≠ null
Then Error ("Función ya declarada")
Else id.entrada := insertarTS (cima (PilaTS), id.lexema)
push (pilaTS, crearTS())
push (pilaDesp, 0) **}**
(L) : T { insertarTipoTS (L.tipo → T.tipo), id.entrada)[1]**}** (L.tipo lo gestionaremos en sus reglas de producción)
; begin P end { pop (PilaTS), pop (PilaDesp)**}**

D → T id ; { If buscarTS (cima (PilaTS), id.lexema) ≠ null
Then Error ("Parámetro ya declarada")
Else id.entrada := insertarTS (cima (PilaTS), id.lexema)
insertarTipoTS (T.tipo, id.entrada)
insertarDespTS (cima (PilaDesp), id.entrada)
cima (PilaDesp) := cima (PilaDesp) + T.ancho [2] **}**

L → T id ; { If buscarTS (cima (PilaTS), id.lexema) ≠ null
Then Error ("Parámetro ya declarada")
Else id.entrada := insertarTS (cima (PilaTS), id.lexema)
insertarTipoTS (T.tipo, id.entrada)
insertarDespTS (cima (PilaDesp), id.entrada)
cima (PilaDesp) := cima (PilaDesp) + T.ancho;
L.tipo := T.tipo **}**

L → L_1 ; L_2 { L.tipo := L_1.tipo x L_2.tipo **}**

T → integer { T.tipo = integer ; T.ancho = 2**}**

T → real { T.tipo = real; T.ancho = 4**}**

S → id := E { id.tipo = buscaTipoTS(id.entrada) ;
If id.tipo = E.tipo
Then S.tipo := tipo_OK
Else S.tipo := tipo_error **}**

Notas

1. Podría parecer que esta función inserta el tipo en la nueva tabla creada, sin embargo insertaTipoTS trabaja de forma inteligente y recorre toda la pila de TS hasta encontrar el identificador correcto.
2. Asumimos que mata la cima de la pila e inserta la nueva cima.
3. buscarTipoTS recorre la pila de TS entera hasta dar con la primera coincidencia.

E → id(A) { If buscaTipoTS (id.lexema)[3] := arg.tipo → t AND arg.tipo = A.tipo
Then E.tipo := t**}**

arg = argumentos de la función
t = valor devuelto por la función

A → E { A.tipo = E.tipo**}**

A → A_1, A_2 { A.tipo := A_1.tipo x A_2.tipo**}**